File System Reliability

Bringing It Together

- Virtual address spaces are mapped onto physical address spaces in physical memory
  - Pages are mapped to frames!
  - Lots of policies to consider: demand paging, page replacement, working set, …
- The heap is part of a process's virtual address space (and is thus paged!)
  - Can be explicitly or automatically managed
- I/O devices are important for system usability.
  - Each one consists of the device, the controller, a bus, and a port
  - One of the most important I/O devices is the disk, which provides stable storage.
- File systems make the disk usable
  - Goals include: efficiency, usability, reliability
  - File system metadata is in a superblock
  - File metadata is in a file header (inode in Linux/Unix)
  - File data is on disk in some format (contiguous, …)
  - File systems demonstrate mechanical sympathy, including through reserved space, block groups, first-free or best fit allocation…)
  - Users organize files using directories
    - Directories are just special files that contain name/file number mappings
  - Programmers access the file system using the API

Reliability and Consistency

Our primary measure of reliability will be *consistency.*

- Consistency: Does my data agree with...itself?

You might argue that this is a pretty low bar to meet.

You'd be right!

But if we can't even guarantee the metadata is consistent, what hope do we have of stronger reliability guarantees?

File System Consistency

File System Consistency

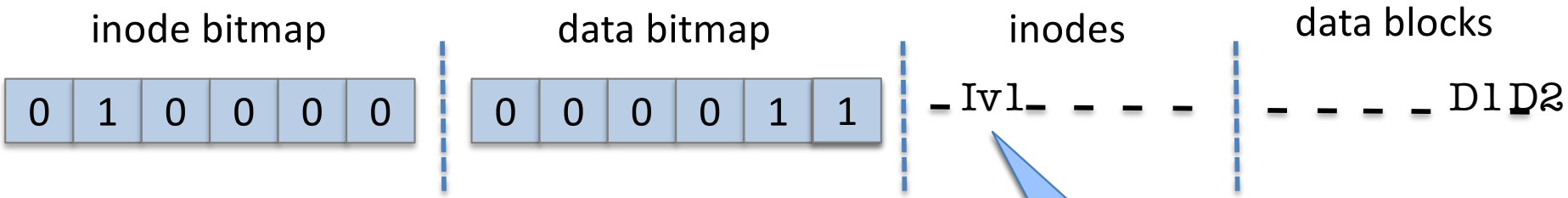
Several file system operations update multiple data structures

- Create a new file
  - Allocate space on disk for header and data
  - Write a new header to disk
  - Add new file to a directory
  - Write data to disk
- Move a file between directories
  - Delete file from old directory
  - Add file to new directory

We need the updated metadata structures to agree with each other.

Example: A Tiny ext3

- 6 blocks, 6 inodes



- Suppose we append a data block to the file
  - add new data block D2
  - update inode
  - update data bitmap

What if a crash or power outage occurs between writes?

owner:	ans
permissions:	read/write
size:	1
pointer:	4
pointer:	null
pointer:	null

Example: A Tiny ext3 Plain Text

- 6 blocks, 6 indoes
- Original data for structures
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 0
  - Inodes: _ [v] _ _ _ _
    - _ means that that spot is empty
    - Information for [v]
      - Owner: ans
      - Permissions: read-only
      - Size: 1
      - Pointer: 4
      - Pointer: null
      - Pointer: null
  - Data blocks: _ _ _ _ D1 _
- Suppose we append a data block to the file
  - Add new data block D2
  - Update inode
  - Update data bitmap
- Updated data for structures
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 1
  - Inodes: _ [v] _ _ _ _
    - _ means that that spot is empty
    - Information for [v]
      - Owner: ans
      - Permissions: read-only
      - Size: 1
      - Pointer: 4
      - Pointer: 5
      - Pointer: null
  - Data blocks: _ _ _ _ D1 D2
- What if a crash or power outage occurs between writes?

What If Only a Single Write Succeeds?

- Just the data block (D2) is written to disk
  - Data is written, but no way to get to it---in fact, D2 still appears as a free block
  - As if write never occurred
- Just the updated inode (Iv2) is written to disk
  - If we follow the pointer, we read garbage
  - *file system inconsistency*: data bitmap says block is free, while inode says it is used. Must be fixed
- Just the updated bitmap is written to disk
  - *file system inconsistency*: data bitmap says data block is used, but no inode points to it.
  - No idea which file should contain the data block!

What If Two Writes Succeed?

- Inode and data bitmap updates succeed
  - File system is consistent
  - But reading new block returns garbage
- Inode and data block updates succeed
  - File system inconsistency
  - Must be fixed
- Data bitmap and data block succeed
  - File system inconsistency
  - No idea which file should contain the data block!

Question

Which is a metadata consistency problem?

A. Null double indirect pointer
B. File created before a crash is missing
C. Data bitmap indicates a block is free that is pointed to by an inode
D. Directory contains a corrupt file name

UNIX and Consistency

Preparing for a Crash

- Metadata
  - If multiple updates are needed, they are performed in a specific order
  - If a crash occurs:
    - Run algorithm to check the file system for consistency (fsck)
    - Check for "*in progress*" operations and fix problems
- Data
  - Blocks are written to disk in any order
  - User programs that care about consistency and reliability store new versions of data in temporary files and replace older version only when user commits

fsck

- UNIX program to check the filesystem for inconsistencies and repair themß.
- Scans all the metadata for inconsistencies
- Fixes any found problems based on known order of disk updates for any operation

Example: File Create

Update Order to Disk

- Write data block
- Update inode
- Update inode bitmap
- Update data bitmap
- Update directory data
- If directory needed another data block:
  - update data bitmap
  - update directory inode

Recovery Plan

- Prior to update of inode bitmap: writes disappear
- Data block referenced in inode, but not in data bitmap: update data bitmap
- File created but not in any directory: place in lost+found

Notice that recovery is possible due to redundancy built into the file system.

This method seems difficult.

and time consuming.

How often do we have to do it?

It gets worse.

How is the file system hiding the latency of the disk?

The Buffer Cache and Writes

- File systems have many data structures that they cache to get good performance
  - Bit map/list of free blocks
  - Directories
  - File headers
  - Indirect blocks
  - Data blocks
- Keeping them accurate is easy if we read
- But what if we write?

Writing to Caches

- Modified data kept in memory can be lost!
- *Write-through (now)*: write changes immediately back to disk
  - Consistent
  - *Slow!* We have to wait for the write to hit the disk and generate an interrupt
- *Write-back (later)*: delay writing the modified data until, for example, the page is replaced in memory
  - Better performance
  - Can cause inconsistencies since the data can be lost in a crash

Cache Updates

- To keep metadata consistent, UNIX uses synchronous write-through
- UNIX uses asynchronous write-back for user data
  - Write-back forced after fixed time intervals (30 sec)
  - Can lose data written within time interval
  - User can also issue the `fsync` command to force the system to send all outstanding writes to disk

Issues

- Ad hoc approach
  - As in, not principled
- Need to get reasoning exactly right
- Write-through caching of metadata leads to poor performance
- Recovery is slooooow: must scan all of the metadata structures
  - Bitmaps, inodes, directories, …

Another Problem

- What if we need multiple file operations to occur as a unit?
  - If you transfer money from one account to another, you need to update the two account files as a unit!
- What if we need *atomicity*?

Solution: *Transactions*

Journaling File Systems

Transactions (Review)

- *Transactions* group actions together so that they are:
  - *atomic*: they all happen or they all don't
  - *serializable*: transactions appear to happen one after the other
  - *durable*: once it happens, it sticks
- Critical sections give us atomicity and serializability, but not durability

Achieving Durability, In Reality (Review)

Reversing changes on disk (*rolling back*) is hard

- How do we know what was there before?

So instead…

- Keep *write-ahead* log on disk of all changes in the transaction
- Once all changes are written to the log, the transaction is committed
  - Which means the word “Commit” is written at the end
  - If a crash happens before the commit, then the log can just be ignored
- W*rite-behind* changes to the disk
  - As in, later on a thread will actually put those changes in their proper spot in the filesystem
  - If a crash occurs during write-behind, the log will still contain the transaction, and the write-behind step can be restarted on restart

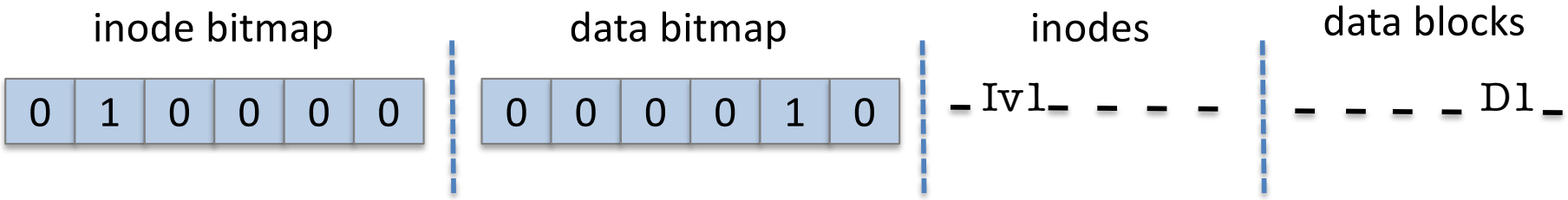
Transactions in File Systems

Most file systems now are *journaling file systems*

- Use transactions for consistency
- Write all *metadata* changes to the *journal*, or transaction log, before updating their place in the file system
  - Including: bitmaps, inodes, directories
  - Not including: update file data, file names
- In the event of a crash, read the journal
  - If journal is empty, then all updates made it to the right place in the file system, do nothing
  - If transaction in journal is not complete (no commit), do nothing
  - If transaction in journal is completely written (committed), apply any changes to the right place in the file system
- Eliminates the need for fsck after a crash

Journaling File Systems: An Example

We start with:



- We want to add a new block to the file
  - Involves updates to the inode and the data bitmap
- Three easy steps
  - Write to the journal 4 blocks: `TxBegin | Iv2 | B2 | Commit`
  - Write the blocks Iv2, B2 to the right spot in the file system
  - Mark the transaction free in the journal
- On crash:
  - No commit in the journal, file system not updated
  - Commit in the journal, replay changes to the file system

Data Journaling: An Example Plain Text

- We start with:
  - Inode bitmap: 0 1 0 0 0 0
  - Data bitmap: 0 0 0 0 1 0
  - Inodes: _ [v] _ _ _ _
  - Data blocks: _ _ _ _ D1 _
- We want to add a new block to the file
- 3 easy steps
  - Write to the log 5 blocks: TxBegin | Iv2 | B2 | D2 | TxEnd
    - Write each record to a block, so it's atomic
  - Write the blocks for Iv2, B2, D2 to the FS proper
  - Mark the transaction free in the journal
- What happens if we crash before the log is updated?
  - No commit, nothing to disk---ignore changes!
- What happens if we crash after the log is updated?
  - Replay changes in log back to disk

Journaling and Write Order

- Issuing the 4 writes to the log `TxBegin | Iv2 | B2 | Commit` sequentially is slow
- Issue at once and transform into a single sequential write
  - What if the write order is: `TxBegin | Iv2 | Commit | B2` ?
  - And what if there is a crash between `Commit` and `B2`?
  - The block where `B2` should be written will be 0s and 1s, but will they be the right 0s and 1s?
- Syntactically, transaction log looks fine, even with nonsense in place of `B2`!
- What do we do?
  - Issuing the 4 writes to the journal sequentially is slow
  - New instruction: `Barrier`
    - Set a `Barrier` before `Commit`
    - Writes before `Barrier` must occur before those after
  - `Commit` must block until other blocks are on disk

Transactions in File Systems

- Advantages:
  - Reliability
  - Asynchronous write-behind
- Disadvantages:
  - All data is written twice!

Question

Where on disk would you put the journal for a journaling file system?

A. Anywhere
B. Outer rim
C. Inner rim
D. Middle
E. Wherever the inodes are

Question

Does journaling prevent you from losing the contents of your files?

A. Yes
B. No
C. It depends

RAID

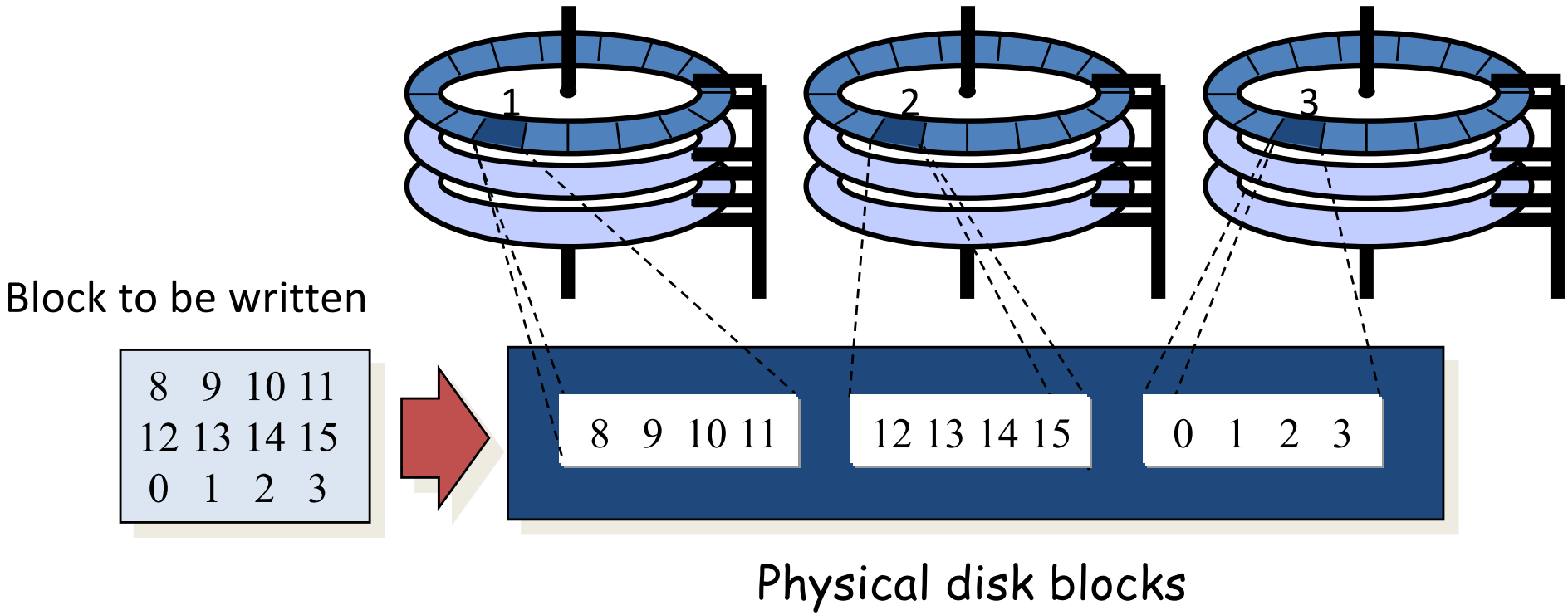
RAID

- *Redundant Array of Inexpensive Disks*
- Disks are cheap, so put many (10s to 100s) of them in one box to increase storage, performance, and availability
- Data plus some redundant information is striped across disks
- Performance and reliability depend on how precisely it is striped
- 5 different levels
  - 0 improves performance
  - 1 improves reliability
  - 3 improves reliability
  - 4 & 5 improve both

RAID-0: Increasing Throughput

Disk striping (RAID-0)

- Blocks broken into sub-blocks that are stored on separate disks
- Higher disk bandwidth (faster reads and writes)
- Reading a block accesses all disks
- Poor reliability
  - Failure of a single disk would cause data loss



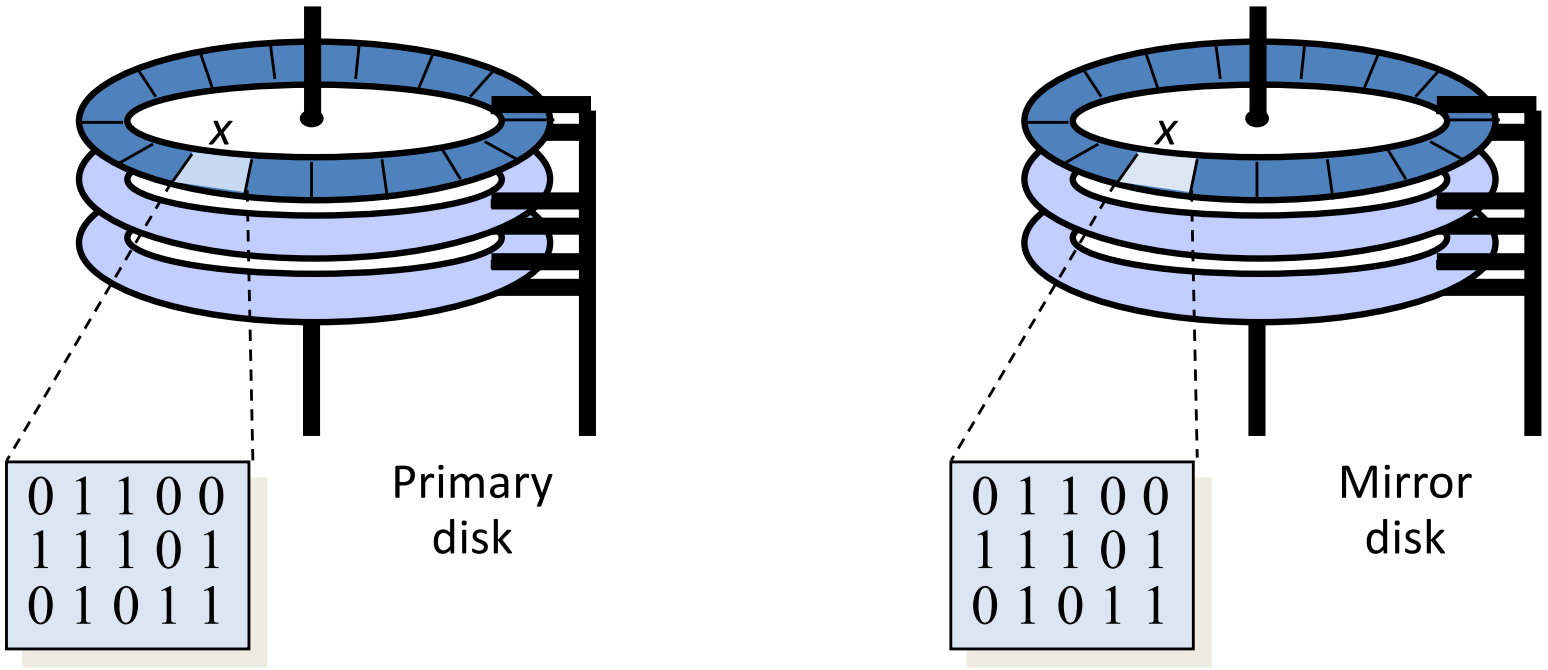
RAID-0: Increasing Throughput Plain Text

- Blocks broken into sub-blocks that are stored on separate disks
- Higher disk bandwidth (faster writes)
- Reading a block accesses all disks
- Poor reliability
  - Failure of a single disk would cause the loss of data
- Example:
  - Block being written holds data: 8 9 10 11 12 13 14 15 0 1 2 3
    - Information 8 9 10 11 stored on disk 1
    - Information 12 13 14 15 stored on disk 2
    - Information 0 1 2 3 stored on disk 3

RAID-1: Mirrored Disks

To increase disk reliability, we must introduce redundancy

- Simple scheme: *Write to both disks, read from either.*
- On failure, use surviving disk



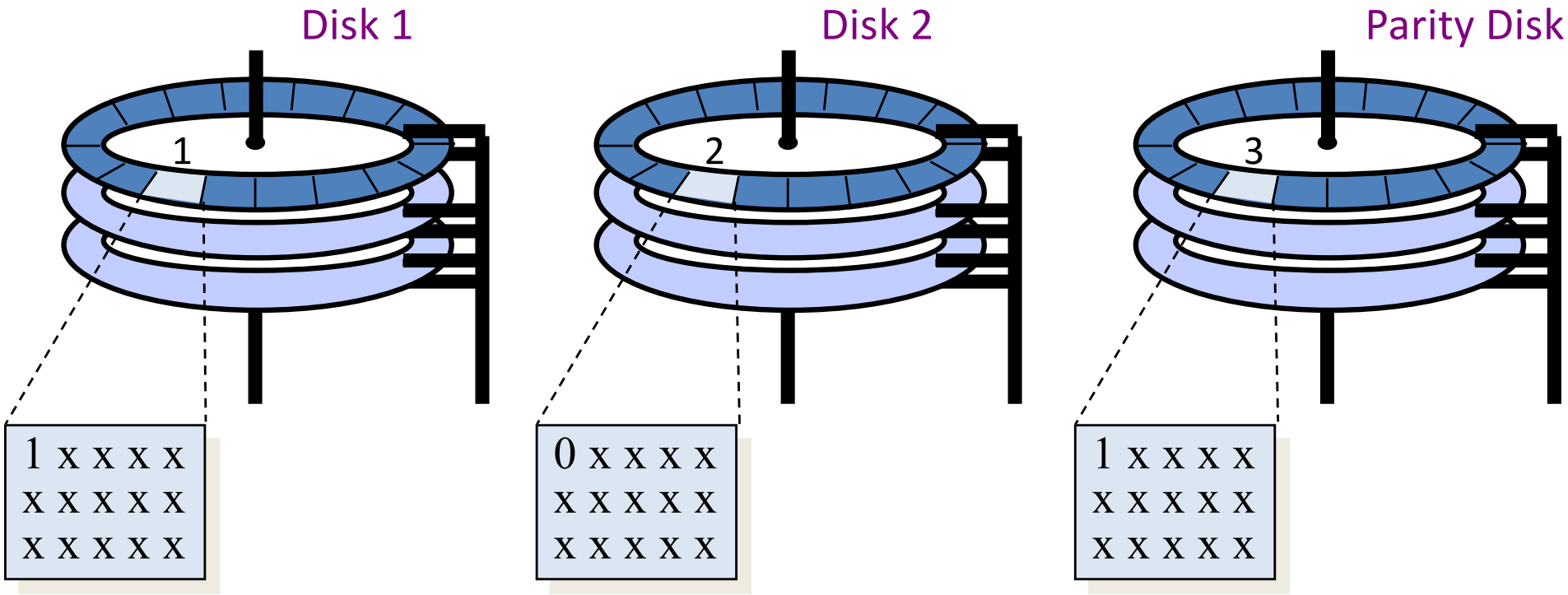
RAID-1: Mirrored Disks
Plain Text

To increase disk reliability, we must introduce redundancy

- Simple scheme: write to both disks, read from either
  - Have 2 disks that each hold all the data for the file system
  - Read from whichever has the head closer to the right spot
- On failure, use surviving disk
- Expensive: have to write each change twice
- Disks marked as “primary” and “mirror”

RAID-3

- Byte-striped with parity
  - Bytes written to same spot on each disk
- Reading a block accesses all data disks
- Writes accesses all data disks plus parity disk
- Disk controller can identify faulty disk
  - Single parity disk can detect and correct errors



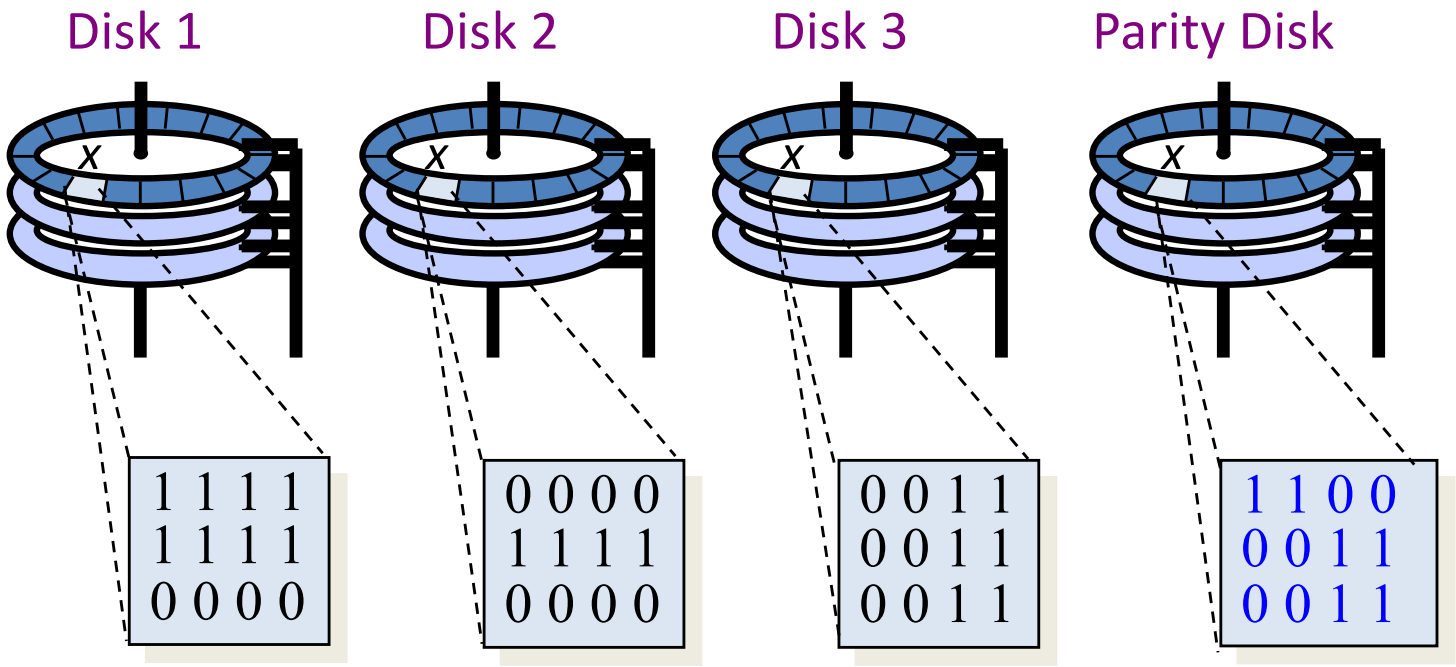
RAID-3: Plain Text

- Byte-striped with parity
  - Bytes written to same spot on each disk
- Reads access all data disks
- Writes access all data disks plus parity disk
- Disk controller can identify faulty disk
  - single parity disk can detect and correct errors
- Example: storing the byte-string 10 and parity bit 1 in RAID-3 system with three disks
  - Store 1 on disk 1, 0 on disk 2 and the parity bit 1 on disk 3
    - parity – evenness/oddness of the bits in the string

RAID-4

- Block striped with parity
  - Blocks written to same spot on each disk
- Reading a block accesses a single disk
- Writing always accesses parity disk
  - Heavy load on parity disk
- Disk controller can identify faulty disk
  - Single parity disk can detect and correct errors

RAID-4 layout:

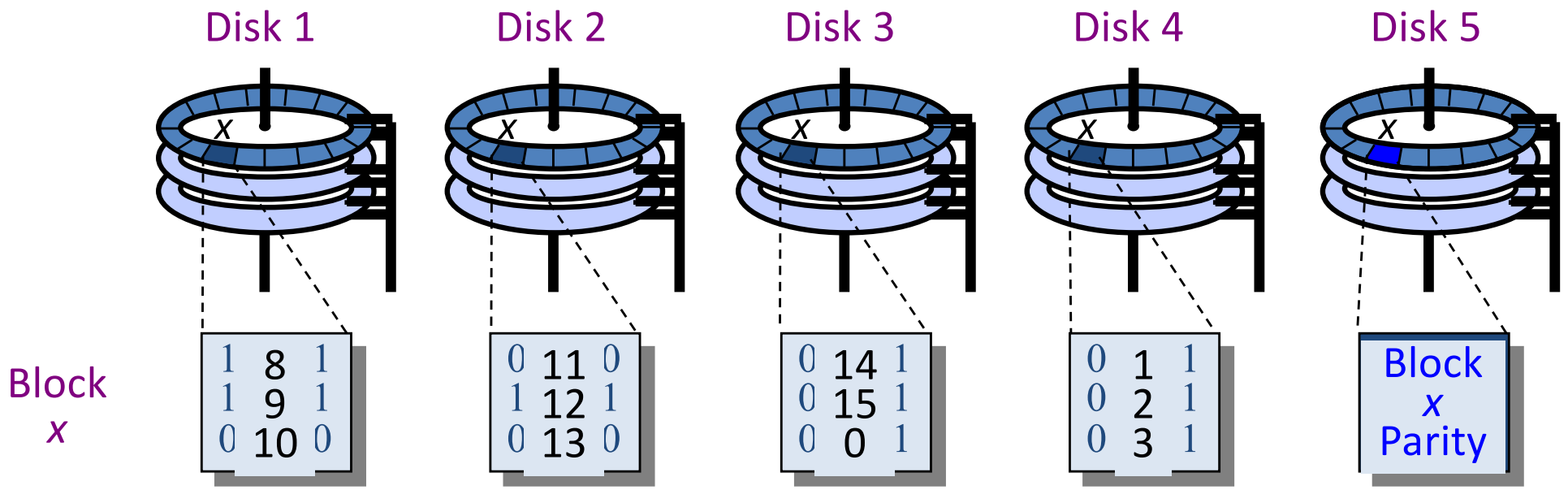


RAID-4: Plain Text

- Block striped with parity
  - Instead of splitting bytes across separate disks you split on block boundaries
  - Blocks written to same spot on each disk
- Combines RAID-0 and RAID-3
  - Reading a block accesses a single block
  - Writing always accesses parity disk
    - Heavy load on parity disk
- Disk controller can identify faulty disk
  - Single parity disk can detect and correct errors

RAID-5

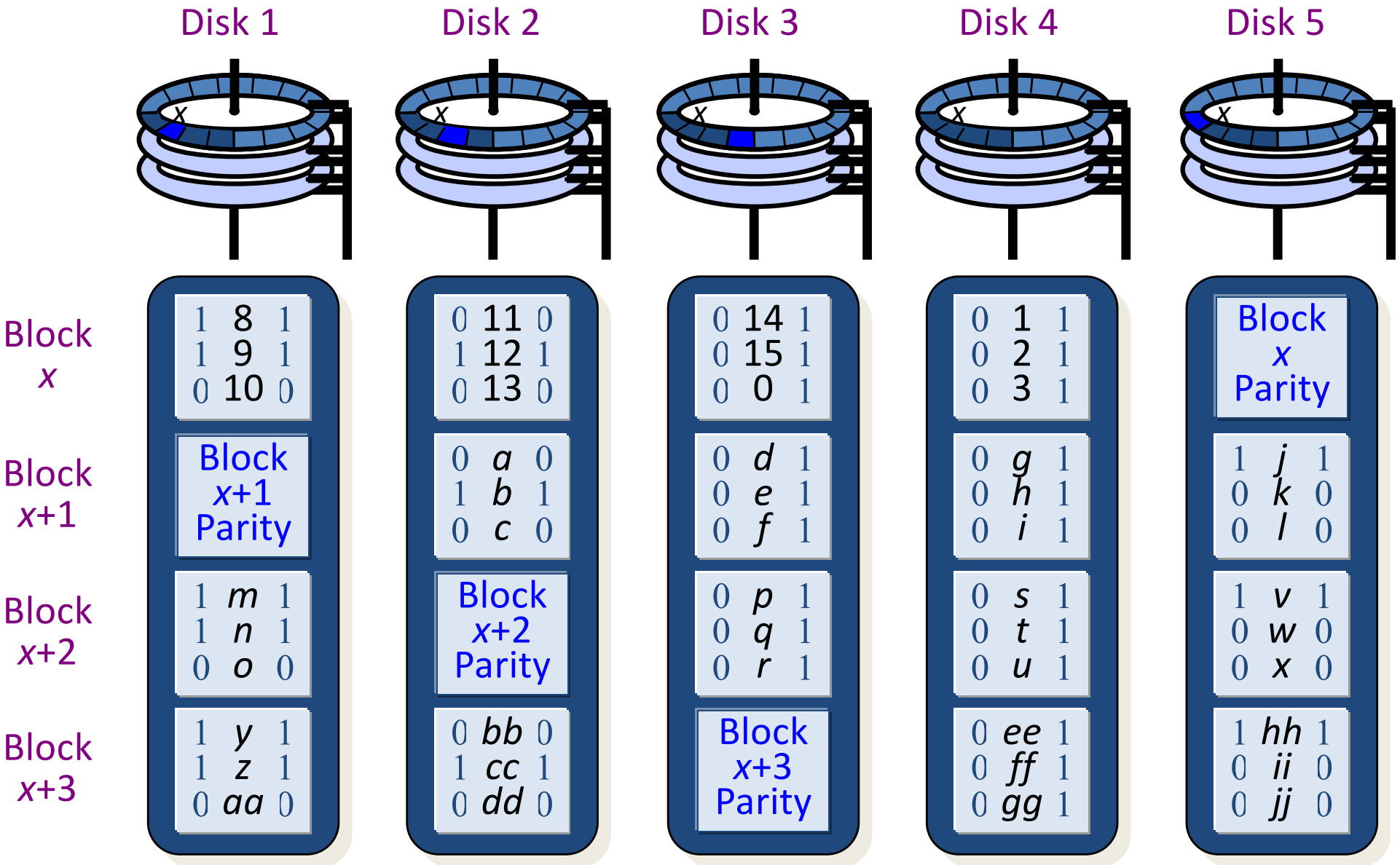
- Block Interleaved Distributed Parity
- No single disk dedicated to parity
- Parity and data distributed across all disks
- Reading a block accesses a single disk



RAID-5: Plain Text

- Block interleaved distributed parity
  - No single disk dedicated to parity
  - Parity and data distributed across all disks
    - So parity bits spread across multiple disks
- Example with 5 disks:
  - 4 blocks written to 4 disks (one to each disk)
  - 5^{th} disk writes parity block
  - Block in same spot on each disk

RAID-5 Example

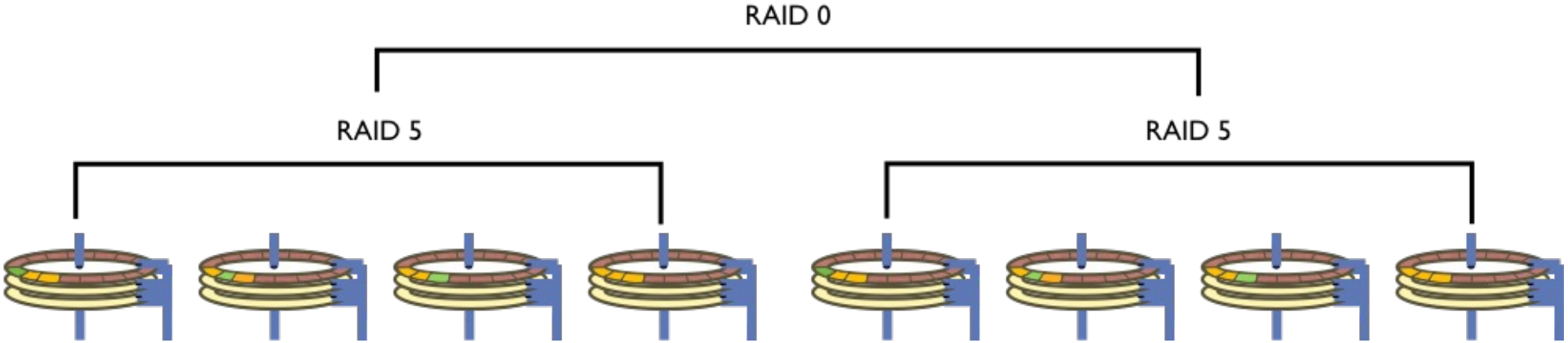


RAID-5 Example: Text Description

- Has 5 separate disks
  - 4 sets of blocks are written
  - In total, 3 blocks of data + 1 parity block are written to each disk
- First set of blocks:
  - 4 data blocks written to disks 1, 2, 3, 4
  - Parity block written to disk 5
- Second set of blocks:
  - 4 data blocks written to disks 2, 3, 4, 5
  - Parity block written to disk 1
- Third set of blocks:
  - 4 data blocks written to disks 3, 4, 5, 1
  - Parity block written to disk 2
- Fourth set of blocks:
  - 4 data blocks written to disks 4, 5, 1, 2
  - Parity block written to disk 3
- Note that in this example, disk 4 does not write a parity block. If the example were to be extended by one data set, it would be disk 4's turn.

RAID-10 and RAID-50

- RAID-10
  - stripes (RAID-0) across reliable logical disks, implemented as mirrored disks (RAID-1)
- RAID-50
  - stripes (RAID-0) across groups of disks with block interleaved distributed parity (RAID-5)



RAID-10 and RAID-50: Plain Text

- RAID-10
  - Stripes (RAID-0) across reliable logical disks, implemented as mirrored disks (RAID-1)
- RAID-50
  - Stripes (RAID-0) across groups of disks with block interleaved distributed parity (RAID-5)
  - Example: Write is striped (RAID-0) to two sets of disks implemented RAID-5.

RAID: Who is in control?

Hardware

- Pros:
  - Tends to be reliable
  - Offloads parity computation from CPU
- Cons
  - Dependent on controller for recovery
  - Serial reconstruction of lost disk

Software

- Pros
  - No additional cost
  - Parallel reconstruction of lost disk
  - Other OS instances might be able to recover
- Cons
  - Ties up CPU to compute parity