Operating Systems

Lecture 13: File System

Department of Computer Science & Technology
Tsinghua University

Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

Basic Concepts

- File System & File
- File Descriptor
- Directory
- File Aliasing
- Types of File System

File System and File

- File system: an OS abstraction for using persistent storage
  - Organizing, manipulating, navigating, accessing, and retrieving data on the persistent storage
- Most computer systems have file systems
  - PCs, servers, laptops
  - iPod, Tivo/set-top-box, cellphones/PDAs
  - Google is made possible by a file system
- File: an OS abstraction for a unit of related data in the file system

File System Functionality

- Allocate disk storage to files
  - Managing file blocks (which blocks belong to which file)
  - Managing free space (which blocks are free)
  - Allocation algorithms (policies)
- Manage the collection of files
  - Locate files and their contents
  - Naming: interface to find files by name
  - Most common: hierarchical file system
  - File system type (different ways to organize files)
- Provide convenience and features
  - Protection: layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc

File and Blocks

- File attributes
  - Name, type, location, size, protection, creator, creation time, last-modified-time, …
- File header
  - On-storage metadata storing information on each file
  - Storing the file attributes
  - Tracking which blocks of the storage belong at which offsets within the logical file structure

Basic Concepts

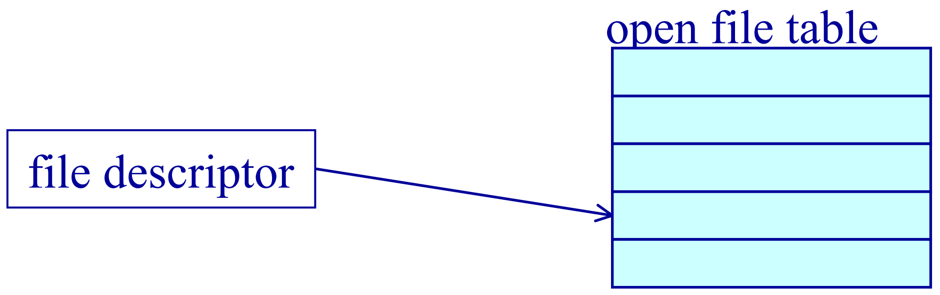
- File System & File
- File Descriptor
- Directory
- File Aliasing
- Types of File System

Open File and File Descriptor

- File use model
  - User program must "open" a file before use

```
f = open(name, flag);
…
… = read(f, …);
…
close(f);
```

- Kernel keeps track of open files for each process
  - OS maintains an open file table per process
  - An open file descriptor is an index into this table



File Descriptor

- Several pieces of data are needed to manage open files:
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it
  - Disk location of the file: cache of data access information
  - Access rights: per-process access mode information

User vs System View of a File

- User’s view:
  - Durable data structures
- At system call interface
  - Collection of bytes (UNIX)
  - Doesn’t matter to system what kind of data structures you want to store on disk!
- OS’s internal view
  - Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
  - Block size 鏐 sector size; in UNIX, block size is 4KB

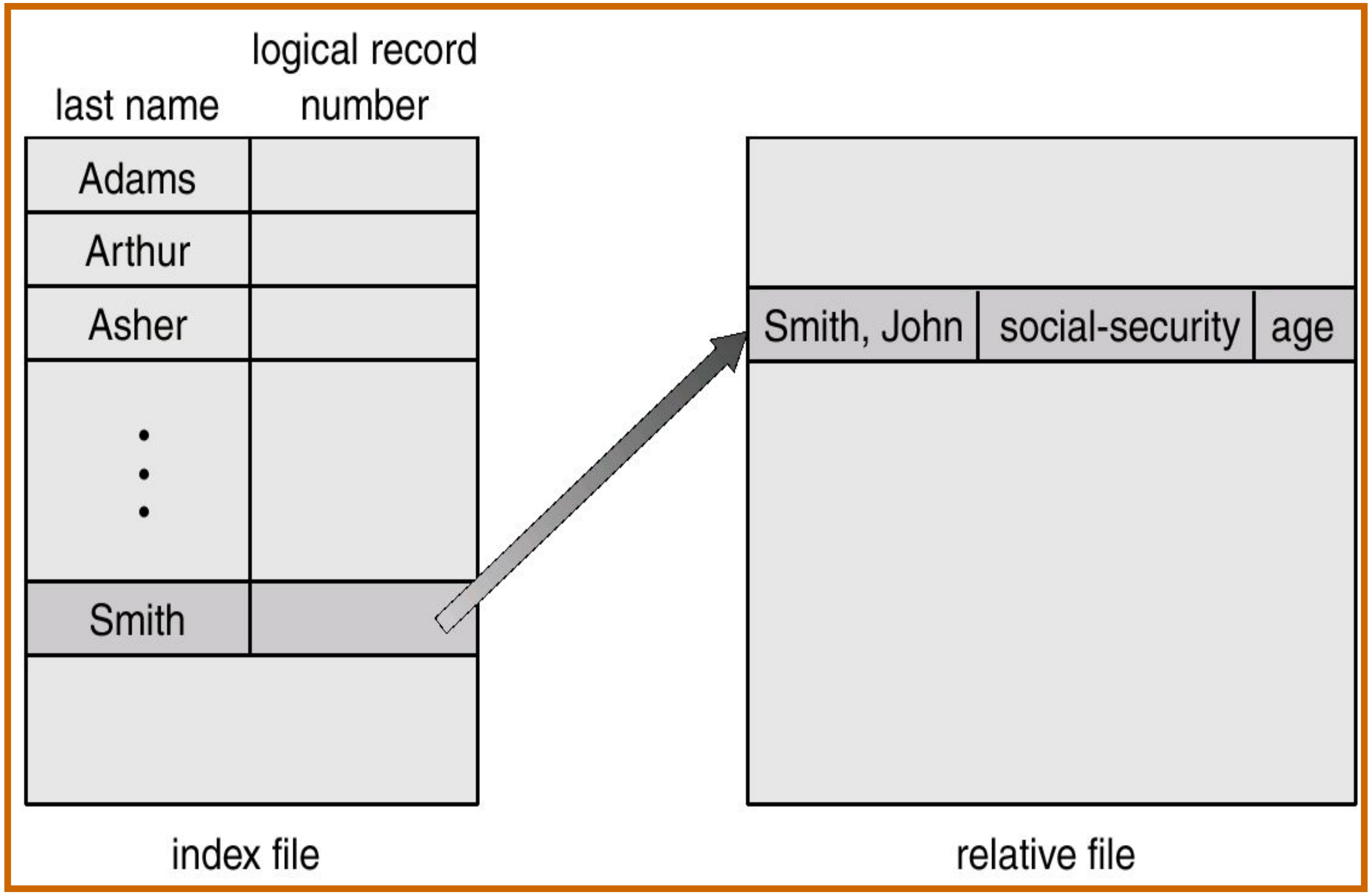
Translating from User to System View

- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, getc(), putc() 鏞 buffers something like 4096 bytes, even if interface is one byte at a time

Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential access: bytes read in order
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don’t want to have to read all bytes to get to the middle of the file
- Content-based Access: by characteristics
  - Many systems don’t provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Example of Index and Relative Files



File Internal Structure

- No structure
  - Sequence of words, bytes
- Simple record structure
  - Lines
  - Fixed length
  - Variable length
- Complex structures
  - Formatted document (e.g., MS Word, PDF)
  - Executable file
  - …

File Sharing and Access Control

- Sharing of files on multi-user systems is desirable
- Access control
  - Who can have what type accesses to what files
  - Types of access: read, write, execute, delete, list, etc.
- Per-file access control list (ACL)
  - <entity, permission>
- Unix model
  - <user|group|world, read|write|execute>
  - User IDs identify users, allowing permissions and protections to be per-user
  - Group IDs allow users to be in groups, permitting group access rights

Consistency Semantics

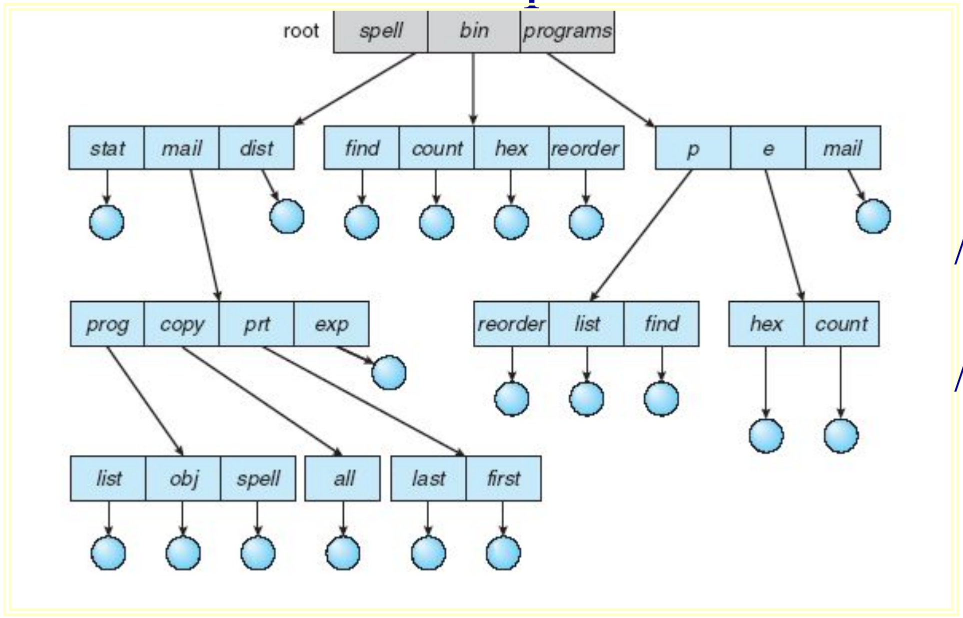
- Specify how multiple users/clients are to access a shared file simultaneously
  - Similar to process synchronization algorithms
  - Less complex due to disk I/O and network latency
- Unix file system (UFS) semantics
  - Writes to an open file are visible immediately to other users of the same open file
  - Sharing file pointer to allow multiple users to read and write concurrently
- Session semantics
  - Writes only visible after the file is closed
- Locking
  - Provided by some OS and file systems

Basic Concepts

- File System & File
- File Descriptor
- Directory
- File Aliasing
- Types of File System

Hierarchical File System

- Files are organized in directories
- Directory is a kind of special files
  - Each contains a <name, pointer to file header> table
- Tree structure for directories and files
  - Some early file systems are flat (single-level directory)
- Hierarchical name space



/spell/mail/prt/first

/programs/p/list

Operations Performed on Directory

- Typical operations
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse a path in the file system
- OS should only allow kernel mode to modify a directory
  - Ensure integrity of the mapping
  - Application programs can read directory (e.g., ls)

Directory Implementation

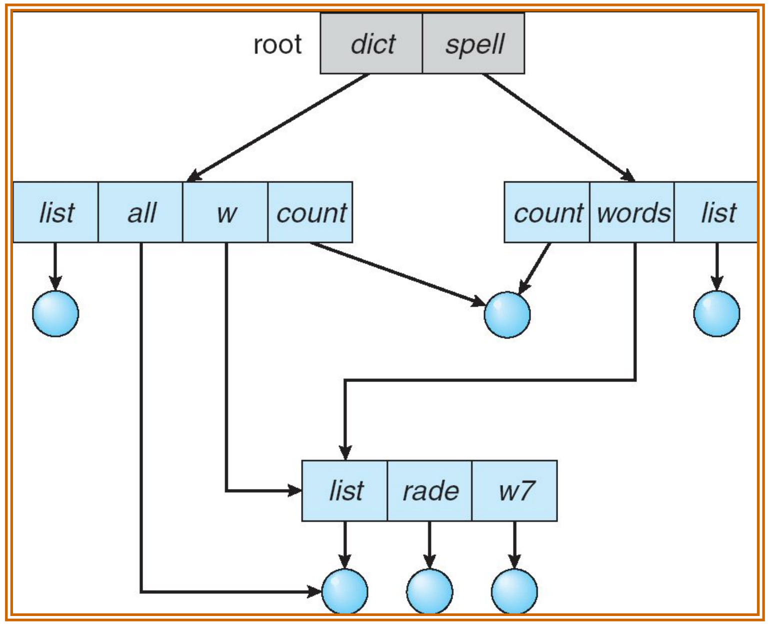
- Linear list of file names with pointer to the data blocks
  - simple to program
  - time-consuming to execute
- Hash Table – linear list with hash data structure
  - decreases directory search time
  - collisions – situations where two file names hash to the same location
  - fixed size

Basic Concepts

- File System & File
- File Descriptor
- Directory
- File Aliasing
- Types of File System

File Aliasing

- Two or more different names referring same file



/dict/count
/spell/count

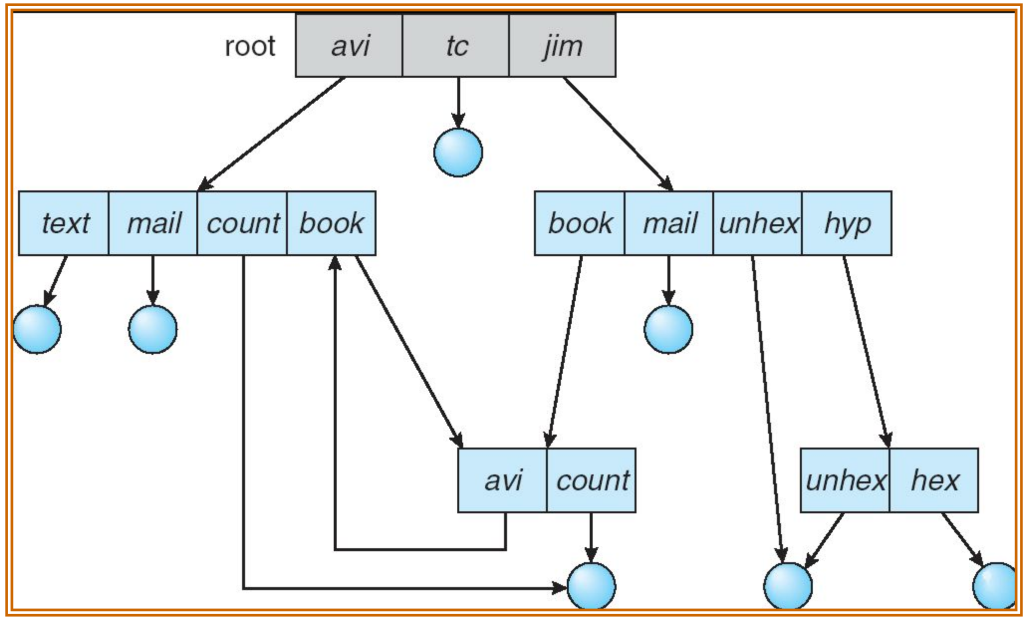
/dict/w/list
/dict/all
/spell/words/list

- Hard Links: multiple directory entries point at the same file
- Soft Links: “shortcut” pointers to other files
  - Implemented by storing the logical name of actual file

The Dangling Pointer Problem in File Aliasing

- What if one delete the file pointed by one name
  - The name alias becomes “dangling pointer”
- Backpointers solution:
  - Each file has a list of backpointers, so we can delete all pointers
  - Backpointers using a daisy chain organization
- Add a level of indirection: directory entry data structure
  - Link – another name (pointer) to an existing file
  - Resolve the link – follow pointer to locate the file

Cycles in Directory



/avi/book/avi/book/avi/book/avi/book/avi/book/avi/book/avi/book/avi/book/avi/…

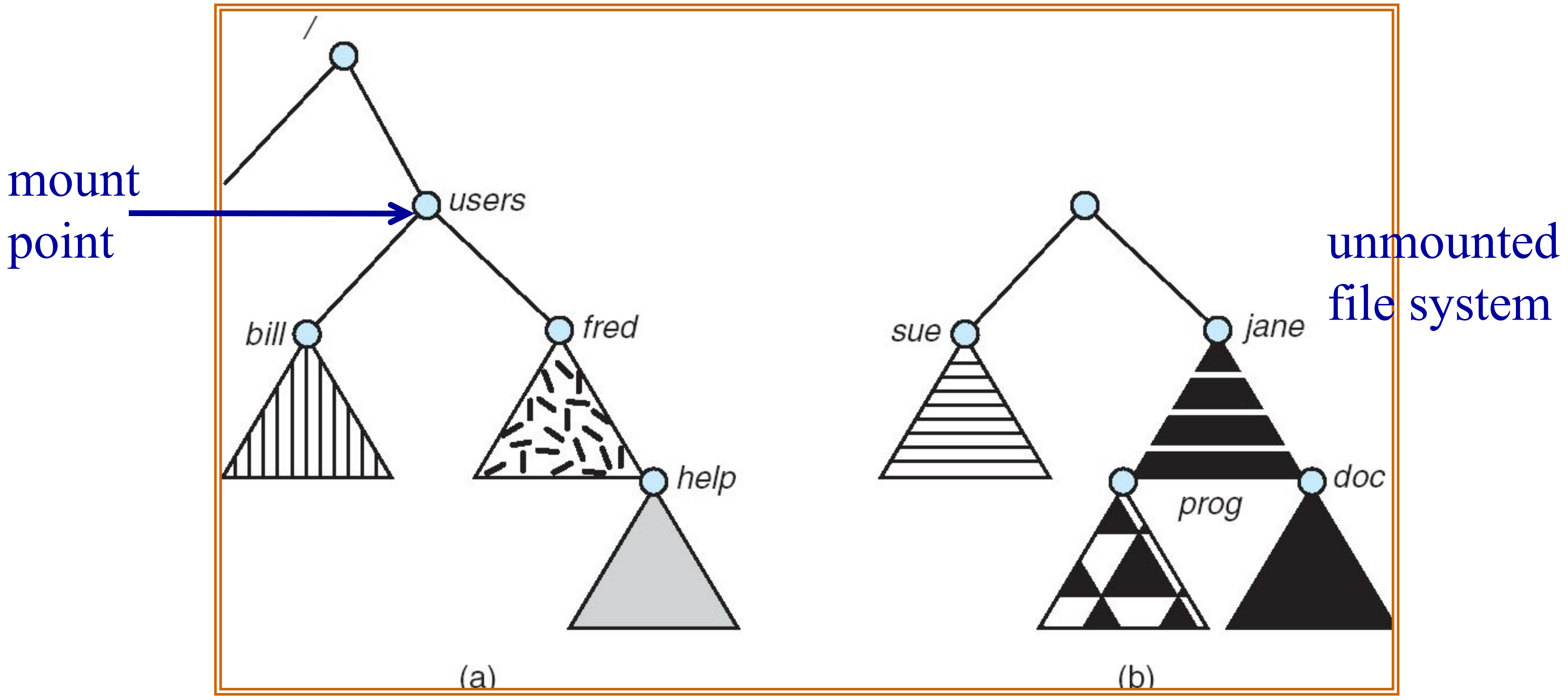
- How do we guarantee no cycles?
  - Allow only links to file not subdirectories
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK
- More practical
  - Limit the number of directories that a path can traverse

Name Resolution: Path Traversal

- Name resolution: the process of converting a logical name into a physical resource (like a file)
  - In file system: file name (path) to actual file
  - Traverse succession of directories until reach target file
- Example: resolving "/bin/ls"
  - Read in file header for root (fixed spot on disk)
  - Read in data block for root; search for "bin" entry
  - Read in file header for "bin"
  - Read in data block for "bin"; search for "ls"
  - Read in file header for "ls"
- Present working directory (PWD)
  - Per-process pointer to a directory for resolving file name
  - Allows user to specify relative path instead of absolute path (say PWD="/bin" can resolve "ls")

File System Mounting

- A file system must be mounted before it can be accessed
- A unmounted file system is mounted at a mount point



Basic Concepts

- File System & File
- File Descriptor
- Directory
- File Aliasing
- Types of File System

Types of File Systems

- Disk file systems
  - Files on a data storage device, like disk.
  - Example: FAT, NTFS, ext2/3, ISO9660, etc.
- Database file systems
  - Files are addressable (resolution) by characteristics
  - Example: WinFS
- Transactional file systems
  - Changes/events to file systems are logged
  - Example: journaling file system
- Network/distributed file systems
  - Example: NFS, SMB, AFS, GFS
- Special/virtual file systems

http://en.wikipedia.org/wiki/Comparison_of_file_systems

Network/Distributed File Systems

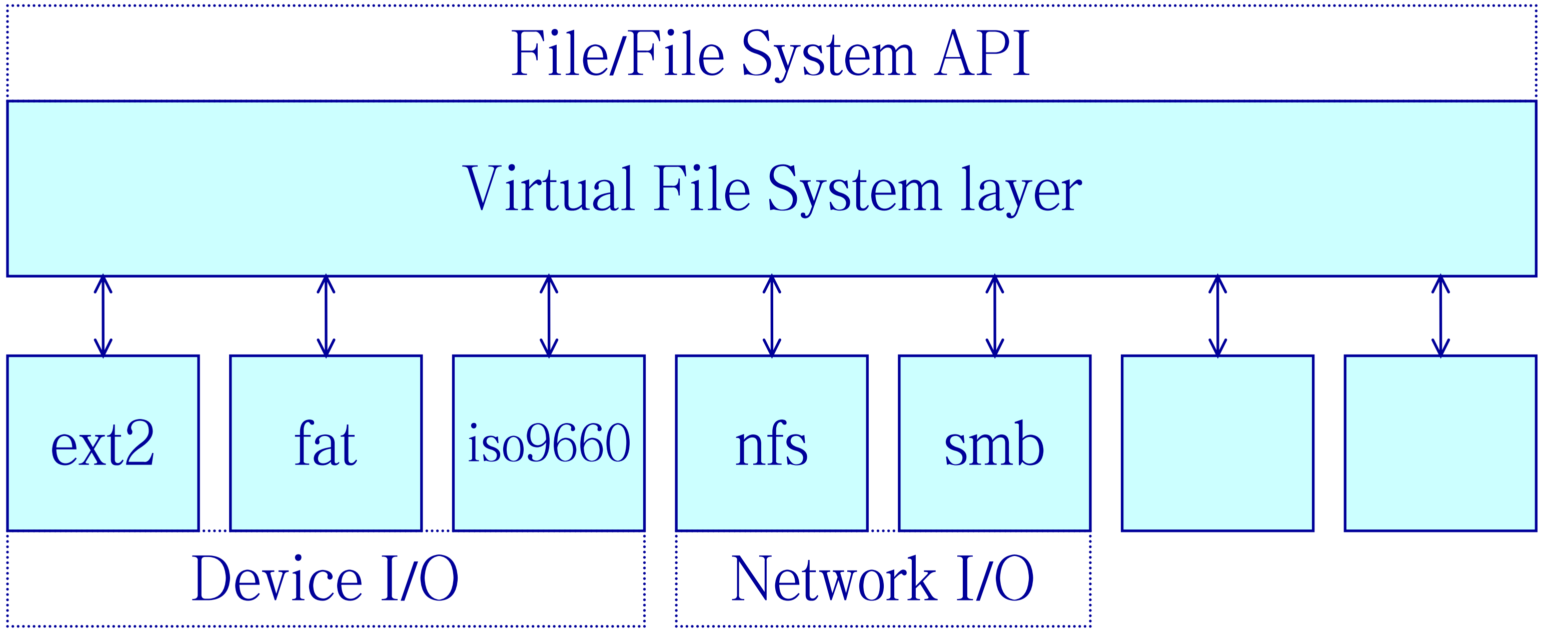
- Files may be shared across a network
  - Files located at remote servers
  - Clients to mount remote file systems from servers
  - Standard OS file calls are translated into remote calls
  - Standard file sharing protocols: NFS for Unix, CIFS for Windows
- Distributed system problems
  - Client and user-on-client identification complicated
  - For example, NFS is insecure
  - Consistency problem
  - Dealing with failure mode
- Truly distributed file systems is still a research
  - Examples: Andrew File System (AFS)

Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

File System Implementation in an OS

- Layering structure
  - Upper layer: virtual (logical) file system
  - Lower layer: specific file system modules



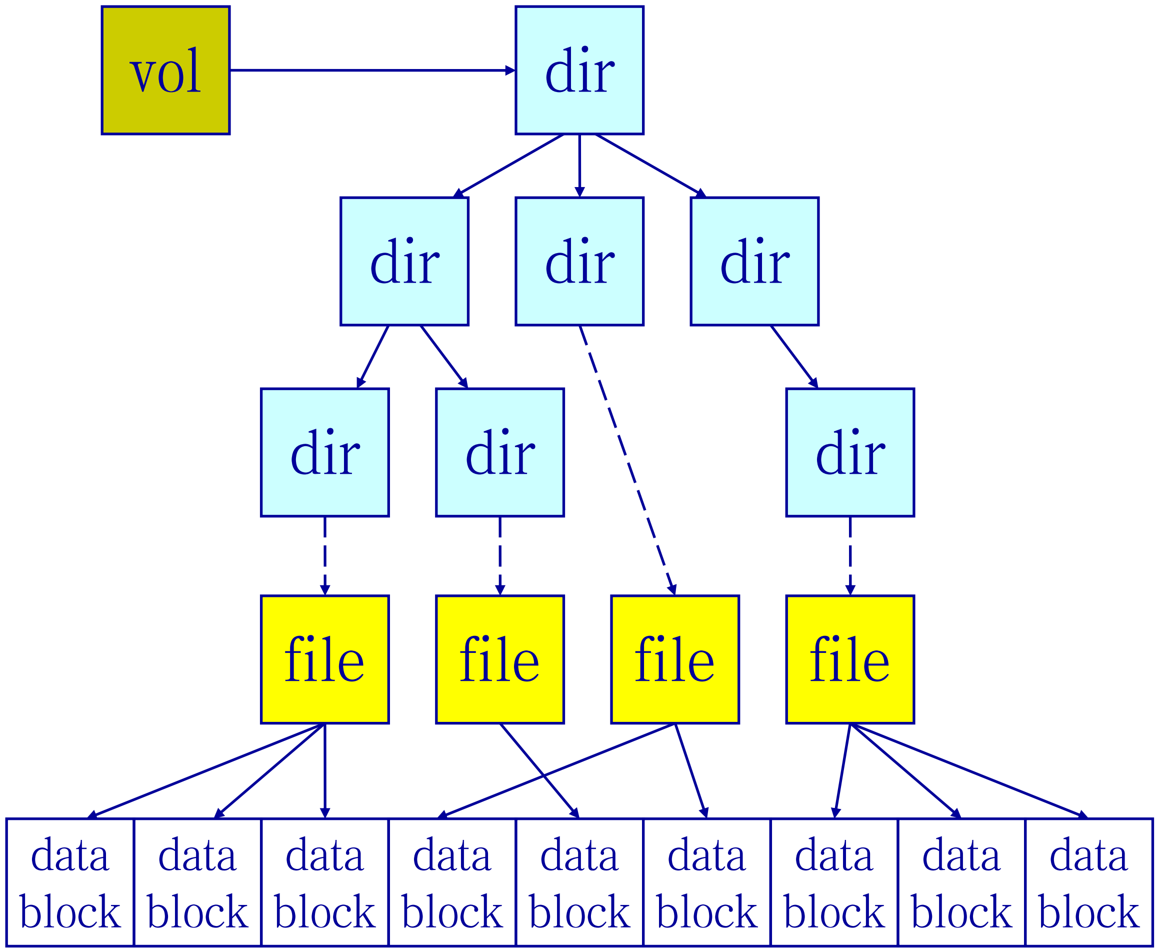
Virtual File System (VFS) Layer

- Purpose
  - Abstraction for all different file system implementations
- Functions
  - Provide the same file and file system interface
  - Manage all file and file system related data structures
  - Routines for efficient lookup, traverse the file system
  - Interact with specific file system modules

File System Basic Data Structures

- Volume Control Block (Unix: “superblock”)
  - One per file system
  - Detail information about the file system
  - # of blocks, block size, free-block count/pointer, etc.
- File Control Block (Unix: “vnode” or “inode”)
  - One per file
  - Detail information about the file
  - Permission, owner, size, data block locations, etc.
- Directory Node (Linux: “dentry”)
  - One per directory entry (directory or file)
  - A tree data structure to encode the directory structure and tree layout
  - Pointer to file control block, parent, list of entries, etc.

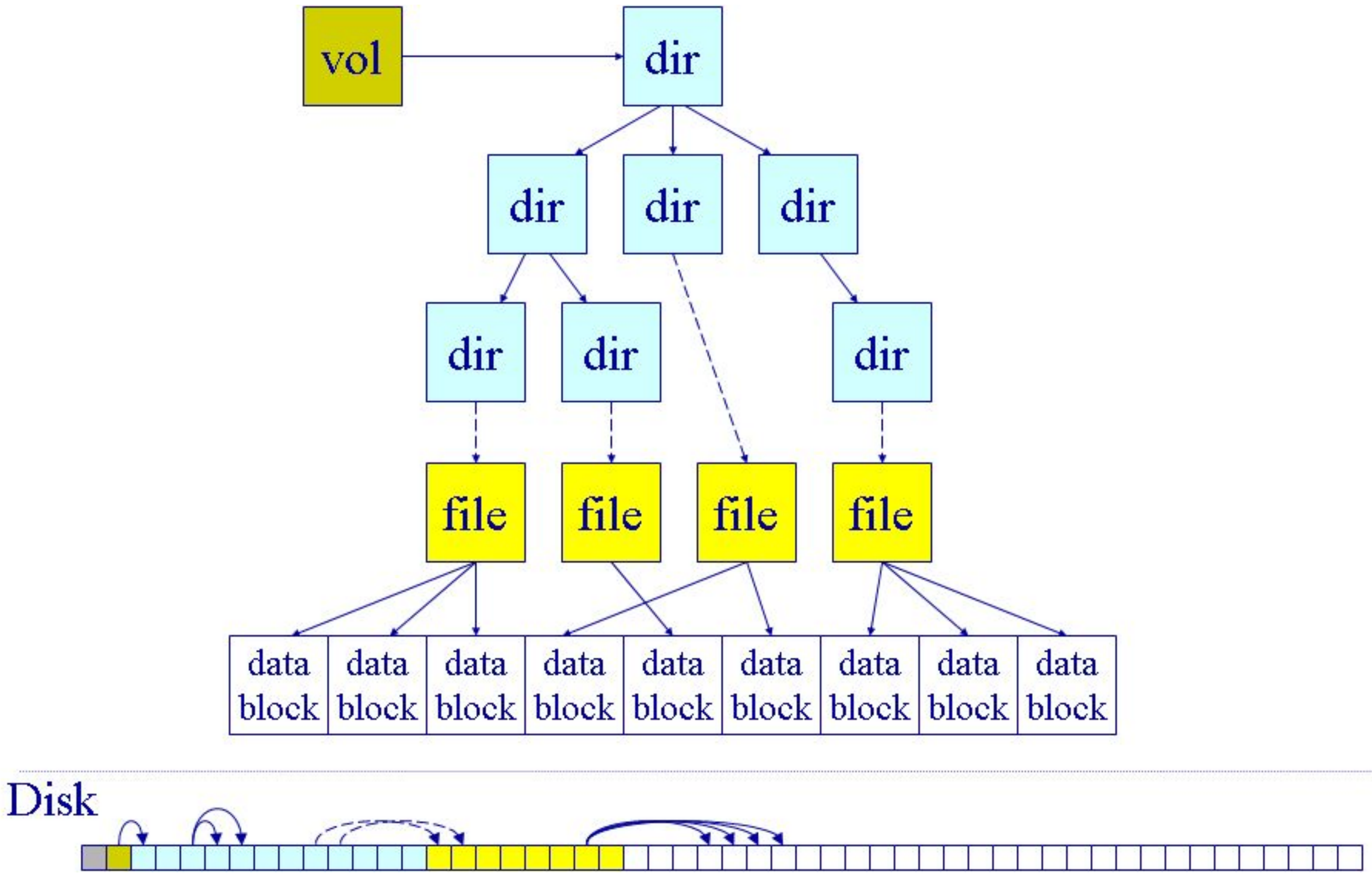
Abstract View



Where are the Data Structure Stored

- File system data structures
  - Volume control block (one per file system)
  - File control block (one per file)
  - Directory node (one per directory entry)
- Persistently stored on the secondary storage
  - In data block(s) allocated in the storage
- Loaded to memory when needed
  - Volume control block: in memory if file system is mounted
  - File control block: if the file is accessed
  - Directory node: during traversal of a file path

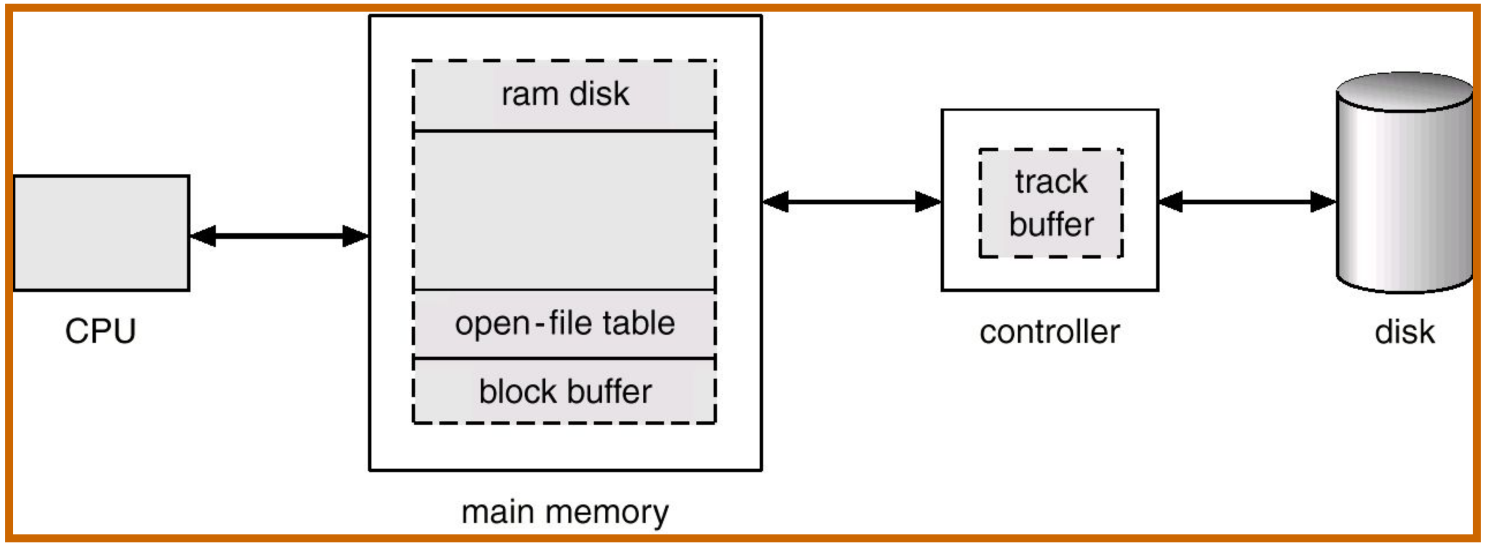
Storage View



Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

Various Disk-Caching Locations

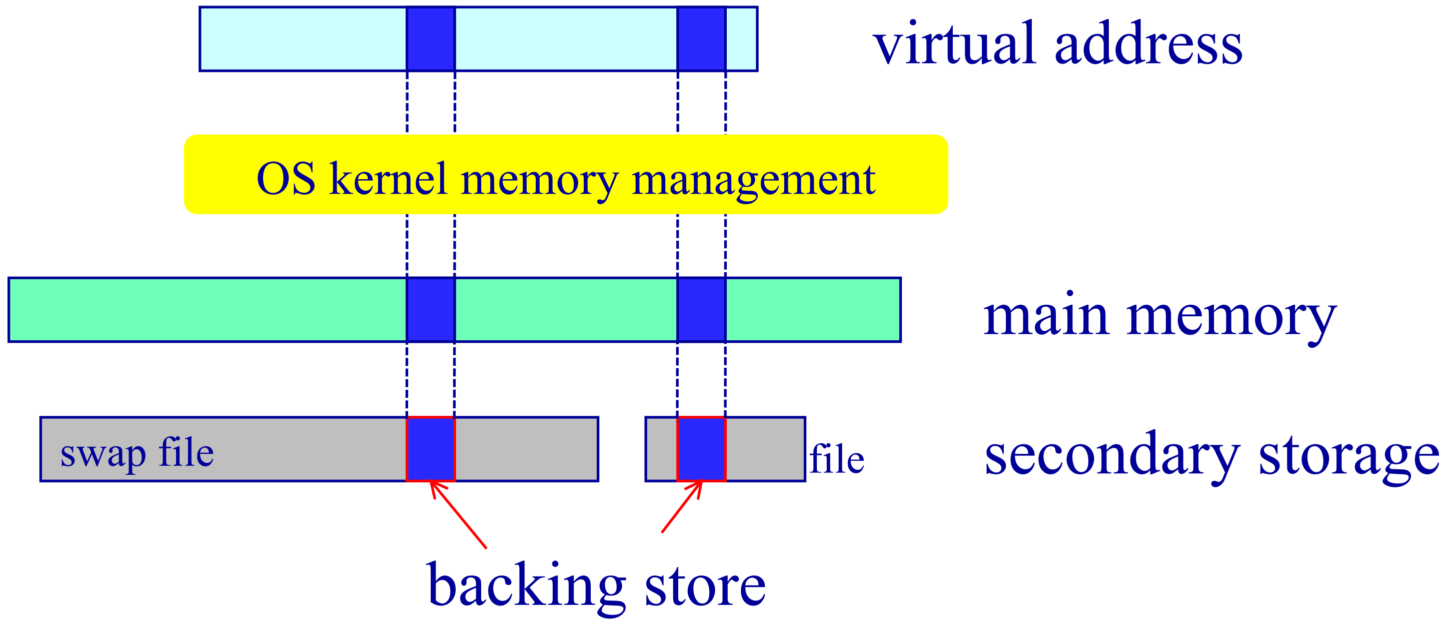


Data Block Caching

- Data blocks are read into memory on-demand
  - To serve a read() operation
  - Read-ahead: prefetch subsequent data blocks
- Data blocks are cached after used
  - Under assumption that they may be used again
  - Writes may be buffered and delayed
- Two methods of caching data block
  - Normal buffer cache
  - Page cache: unified caching for data blocks and memory pages

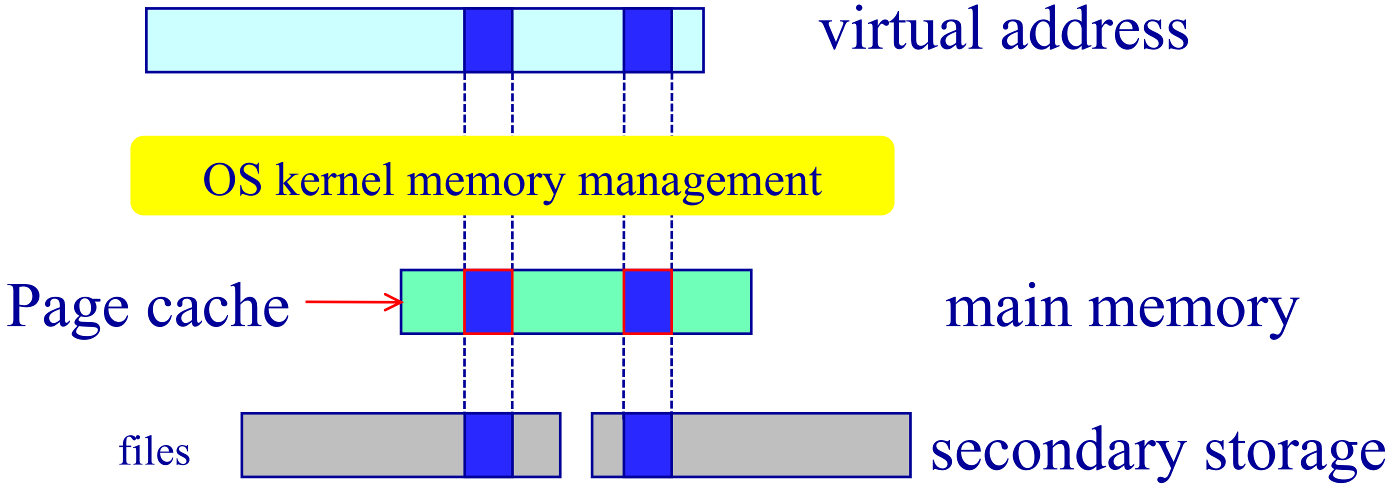
Remember Demanded Paging Memory Model?

- Demand paging
  - Bring a page into memory only when it is needed
- Backing store
  - A page (in virtual address space) can be mapped to a location in a file (in secondary storage)

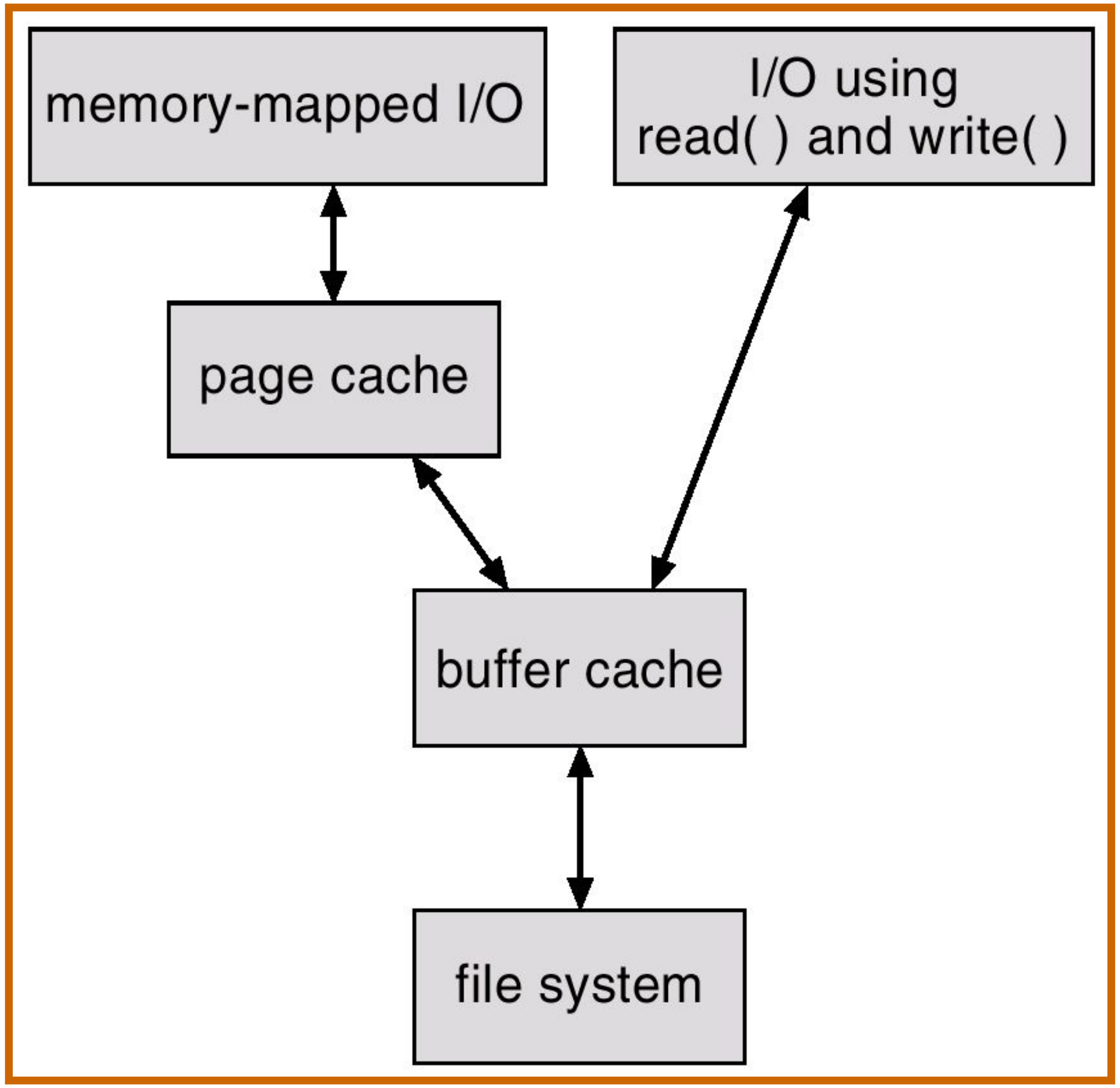


Page Cache

- Page cache for file data blocks
  - A file data block is mapped to a page in virtual memory
  - File read/write op is translated to memory access
  - May cause page-fault and/or set the page dirty
  - Issue: page replacement – taken from processes or file page cache?

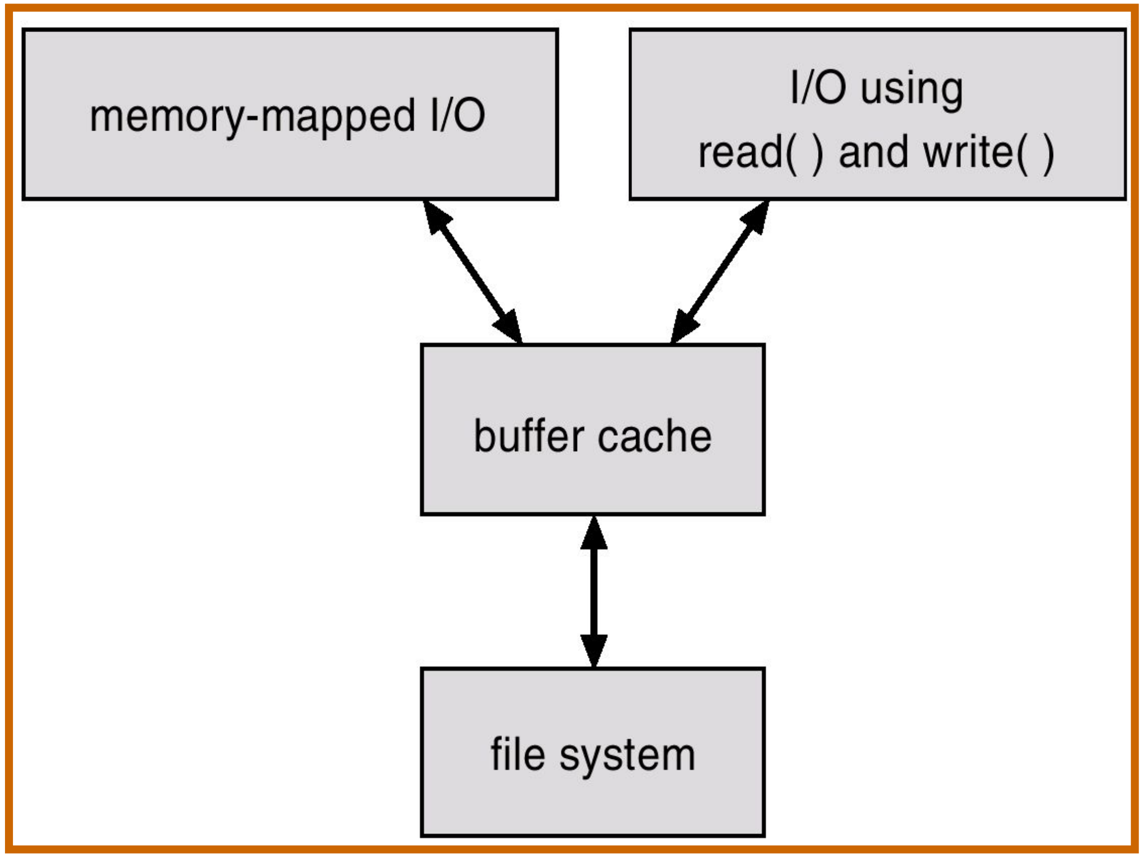


I/O Without a Unified Buffer Cache



Unified Buffer Cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.



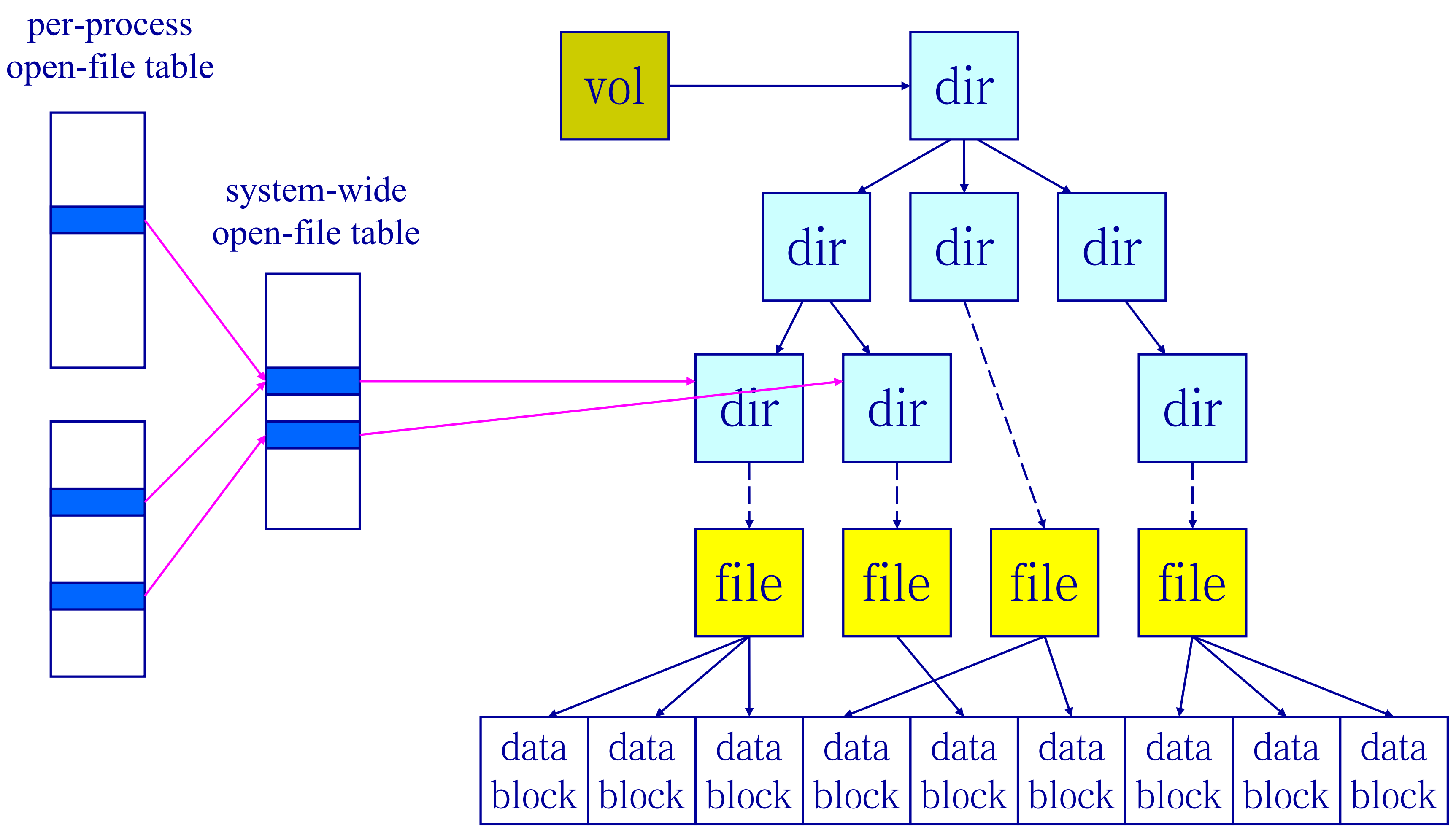
Outline（part2）

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

File System Data Structures for Open Files

- Open file descriptor
  - One per open file
  - Information about the file status
  - Directory entry, current file pointer, set of file ops, etc.
- Open file tables
  - One per process
  - One system-wide
  - Each volume control block should keep a list too
  - So that it wouldn’t dismount if still open file(s)

Open-File Tables



Open File Locking

- Provided by some operating systems and file systems
- Mediates access to a file
- Mandatory or advisory:
  - **Mandatory** – access is denied depending on locks held and requested
  - **Advisory** – processes can find status of locks and decide what to do

Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

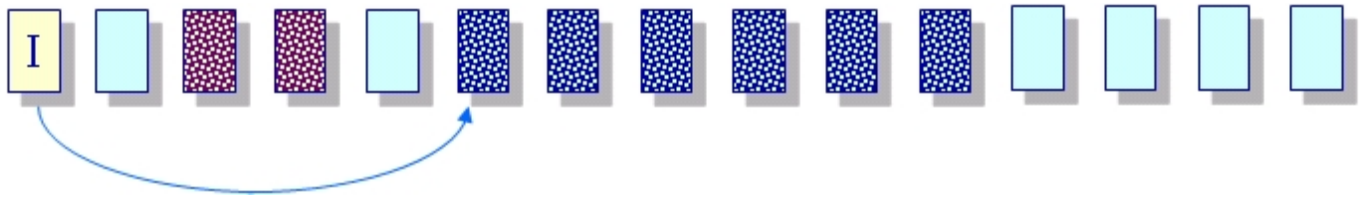
Use Pattern

- Most files are small.
  - Need strong support for small files.
  - Block size can’t be too big.
- Some files are very large.
  - Must allow large files (64-bit file offsets).
  - Large file access should be reasonably efficient.

File Allocation

- How to allocate data blocks to each file
- Allocation methods
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation
- Metrics
  - Efficiency: e.g., storage utilization (external fragmentation)
  - Performance: e.g., access speed

Contiguous Allocation

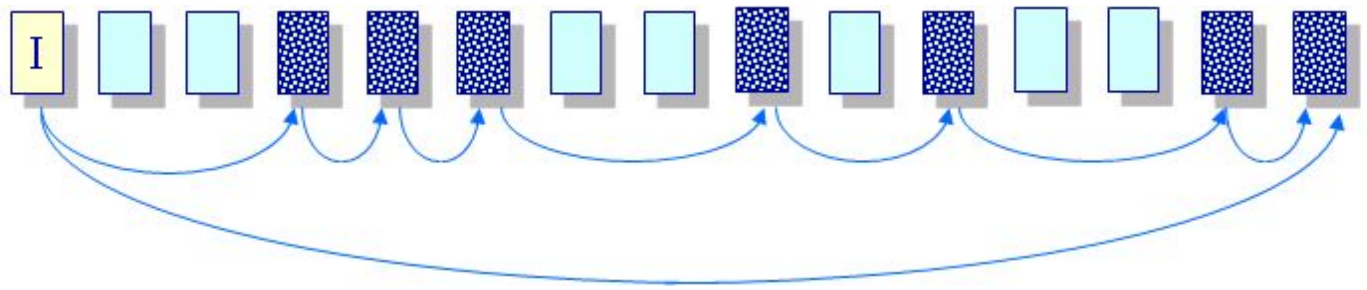


- File header specifies starting block & length
- Placement/Allocation policies
  - First-fit, best-fit, ...

- Pluses
  - Best file read performance
  - Efficient sequential & random access

- Minuses
  - Fragmentation!
  - Problems with file growth
    - Pre-allocation?
    - On-demand allocation?

Linked Allocation

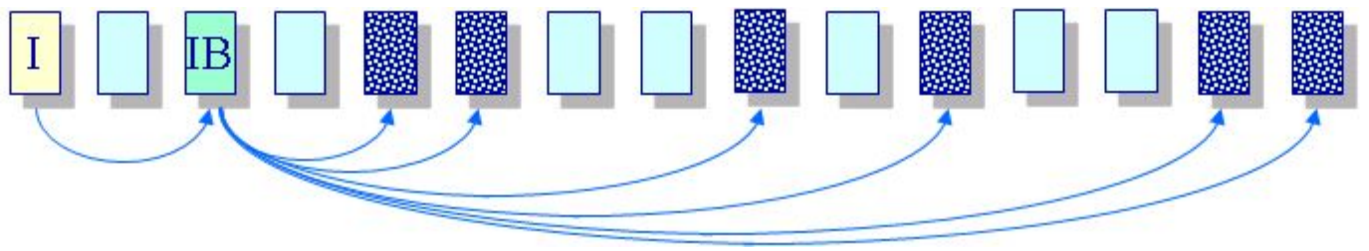


- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks

u Pluses
  - Easy to create, grow & shrink files
  - No fragmentation

鐙 Minuses
  - Impossible to do true random access
  - Reliability
    - Break one link in the chain and...

Indexed Allocation

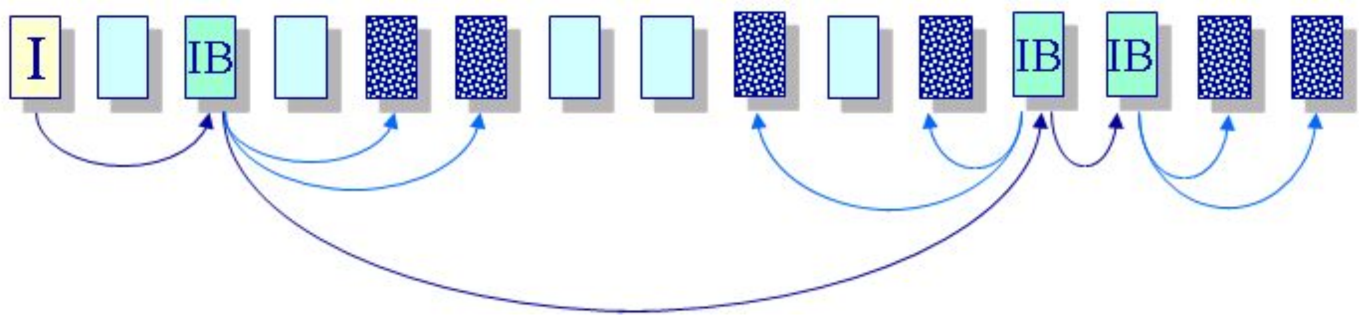


- Create a non-data block for each file called the index block
  - A list of pointers to file blocks
- File header contains the index block

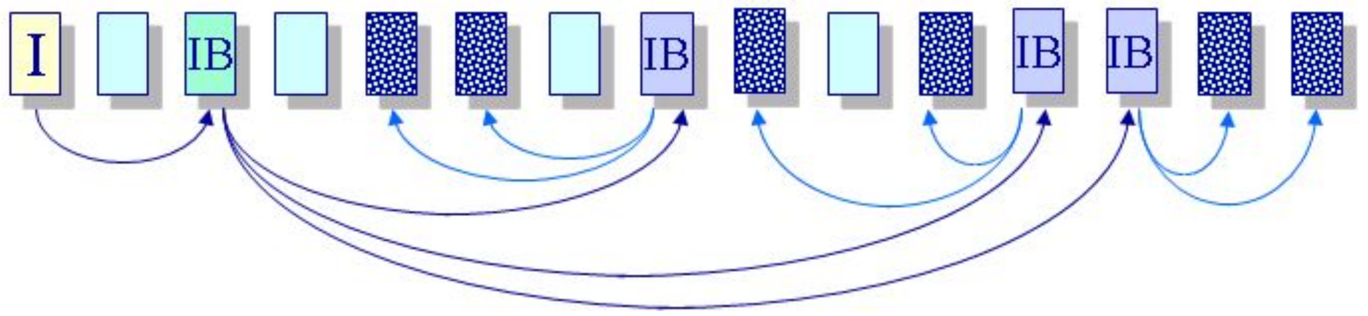
- Pluses
  - Easy to create, grow & shrink files
  - No fragmentation
  - Supports direct access
- Minuses
  - Overhead of storing index when files are small
  - How to handle large files?

Indexed Allocation for Large Files

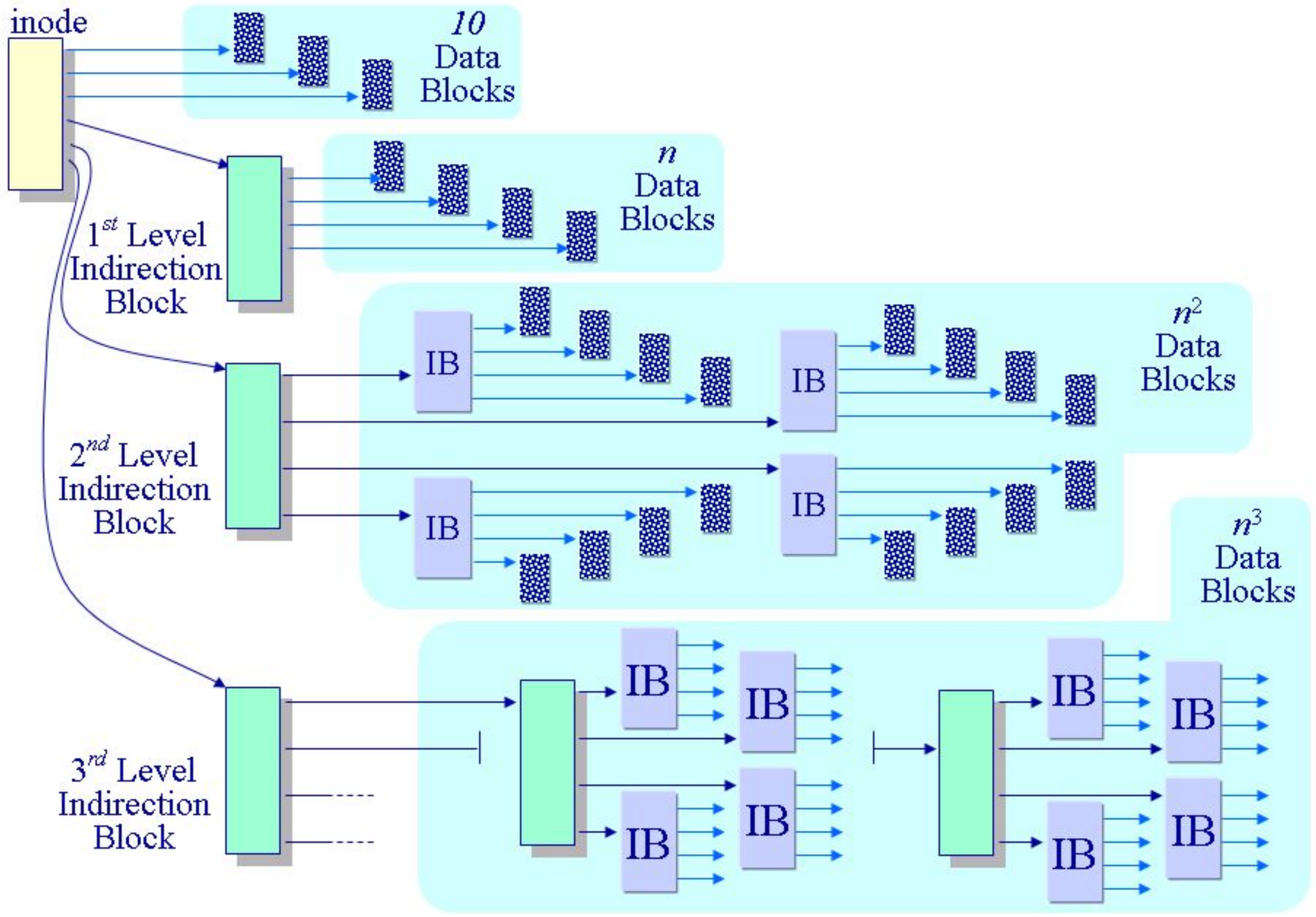
- Linked index blocks (IB+IB+…)



- Multilevel index blocks (IB*IB*…)



Multi-level Indexed Allocation in UNIX



Multi-level Indexed Allocation in UNIX

- File header contains 13 pointers
  - 10 pointes to data blocks;
  - 11th pointer → indirect block;
  - 12th pointer → doubly-indirect block;
  - 13th pointer → triply-indirect block
- Implications
  - Upper limit on file size
  - Blocks are allocated dynamically, files can easily expand
  - Small files are cheap
  - Allocate indirect blocks only for large files, and large files require a lot of seek to access indirect blocks

Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

Free-Space List

- Keep track of all unallocated blocks in the storage
- Where is free-space list stored?
- What is a good data structure for free-space list?

Free-Space List: Bit Map

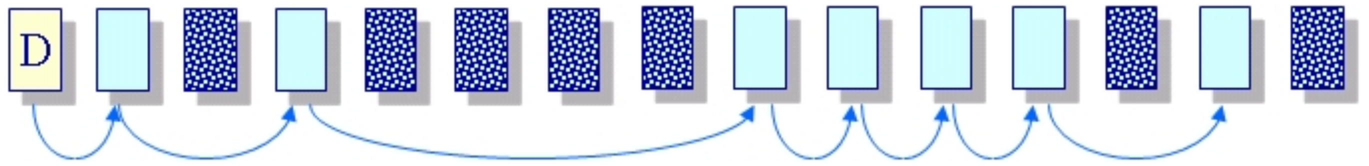
- Represent the list of free blocks as a bit map:
  - 11111111111111001110101011101111...
  - If bit i = 0 then block i is free, otherwise it is allocated
- Simple to use but this can be a big vector:
  - 160GB disk -> 40M blocks -> 5MB worth of bits
  - However, if free sectors are uniformly distributed across the disk then the expected number of bits that must be scanned before finding a “0” is n/r, where
    - n = total number of blocks on the disk
    - r = number of free blocks
  - If a disk is 90% full, then the average number of bits to be scanned is 10, independent of the size of the disk

Free-Space List: Bit Map (Cont.)

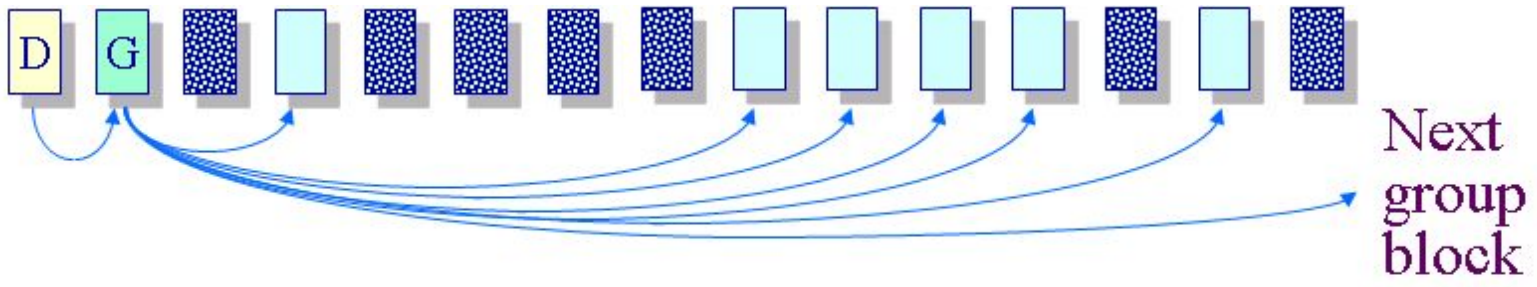
- Need to protect:
  - Pointer to free list
  - Bit map
    - Must be kept on disk
    - Copy in memory and disk may differ.
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk.
  - Solution:
    - Set bit[*i*] = 1 in disk.
    - Allocate block[*i*]
    - Set bit[*i*] = 1 in memory ?

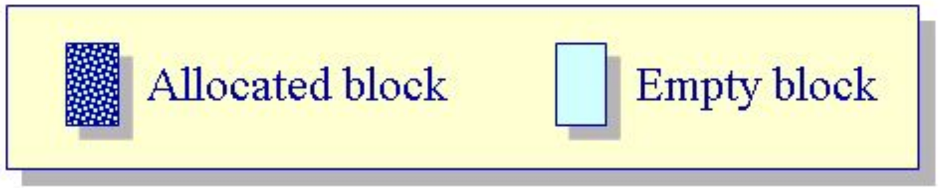
Other Free List Representations

- linked lists



- Grouped lists



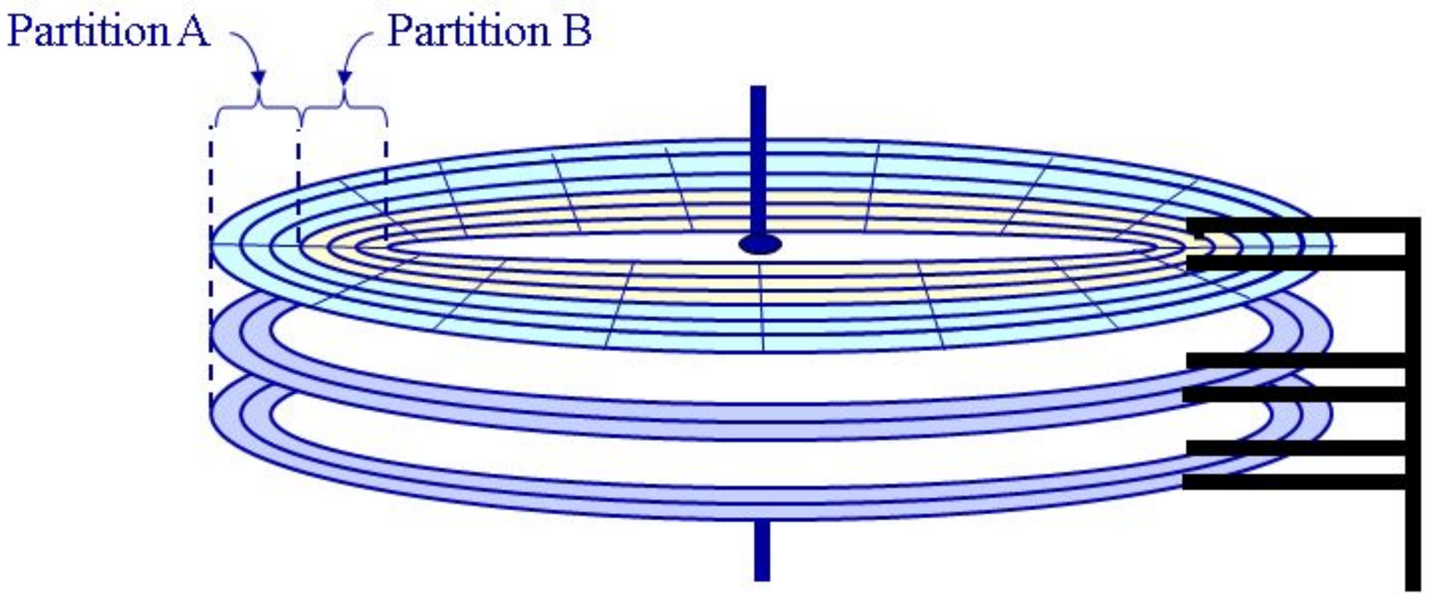


Outline

- Basic Concepts
- Virtual File System
- Data Block Caching
- Data Structures for Open Files
- File Allocation
- Free-Space List
- Management of Multiple Disks – RAID

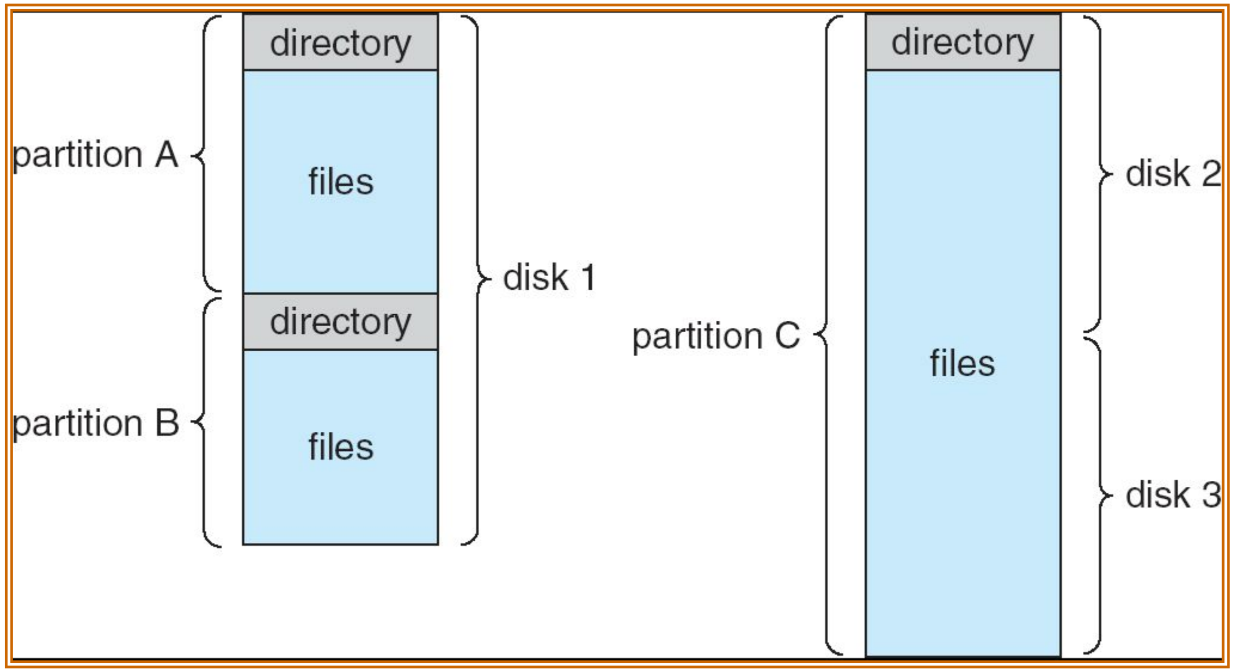
Disk Partitioning for Performance

- Disks are typically partitioned to minimize the largest possible seek time
  - A partition is a collection of cylinders
  - Each partition is a logically separate disk



A Typical Disk File-System Organization

- Partition: a division of hard disk to apply OS-specific formatting
- Volume: a single accessible storage area with a single instance of a filesystem
  - Typically resident on a single partition of a hard disk

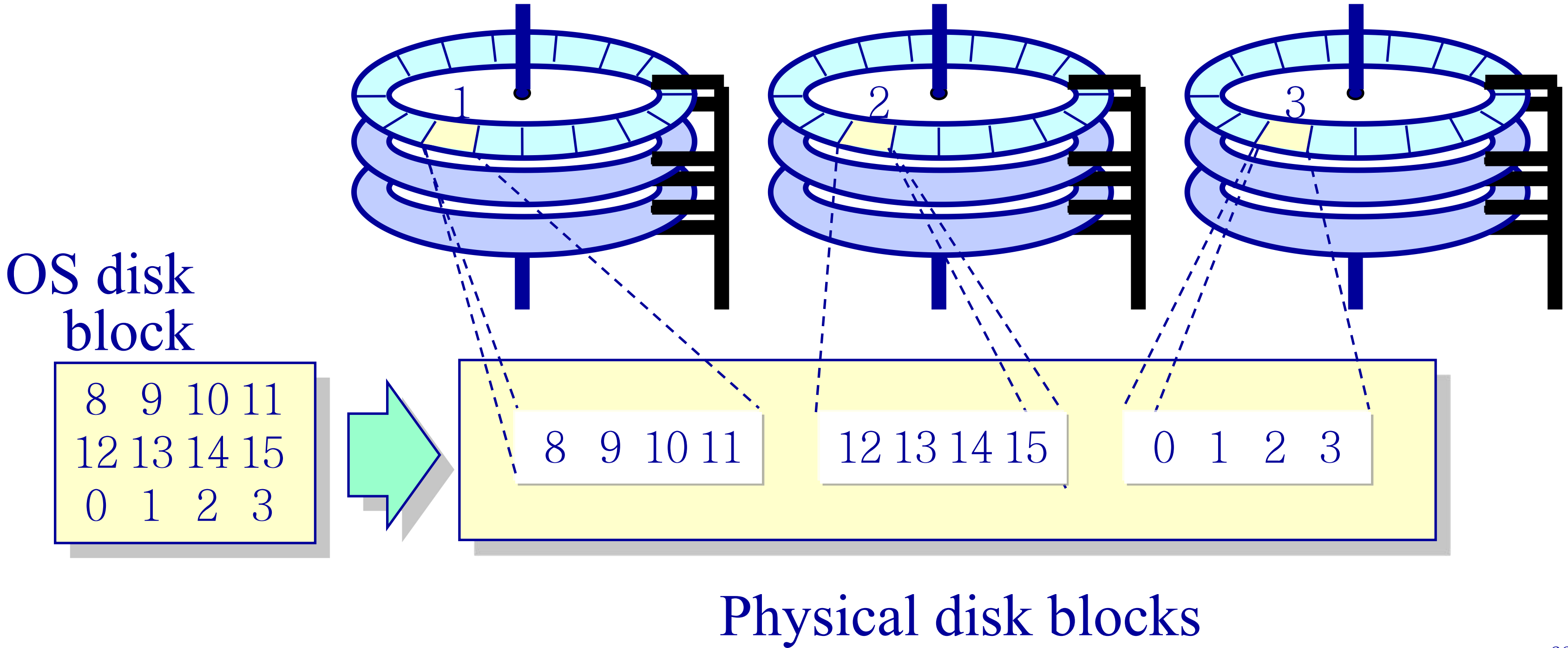


Management of Multiple Disks

- Use multiple parallel disks to increase
  - Throughput (through parallelism)
  - Reliability and availability (through redundancy)
- RAID - Redundant Array of Inexpensive Disks
  - A variety of disk-organization techniques
  - RAID levels: different RAID scheme (e.g., RAID-0, RAID-1, RAID-5)
- Implementation
  - In OS kernel: storage/volume management
  - In hardware RAID controller (I/O)

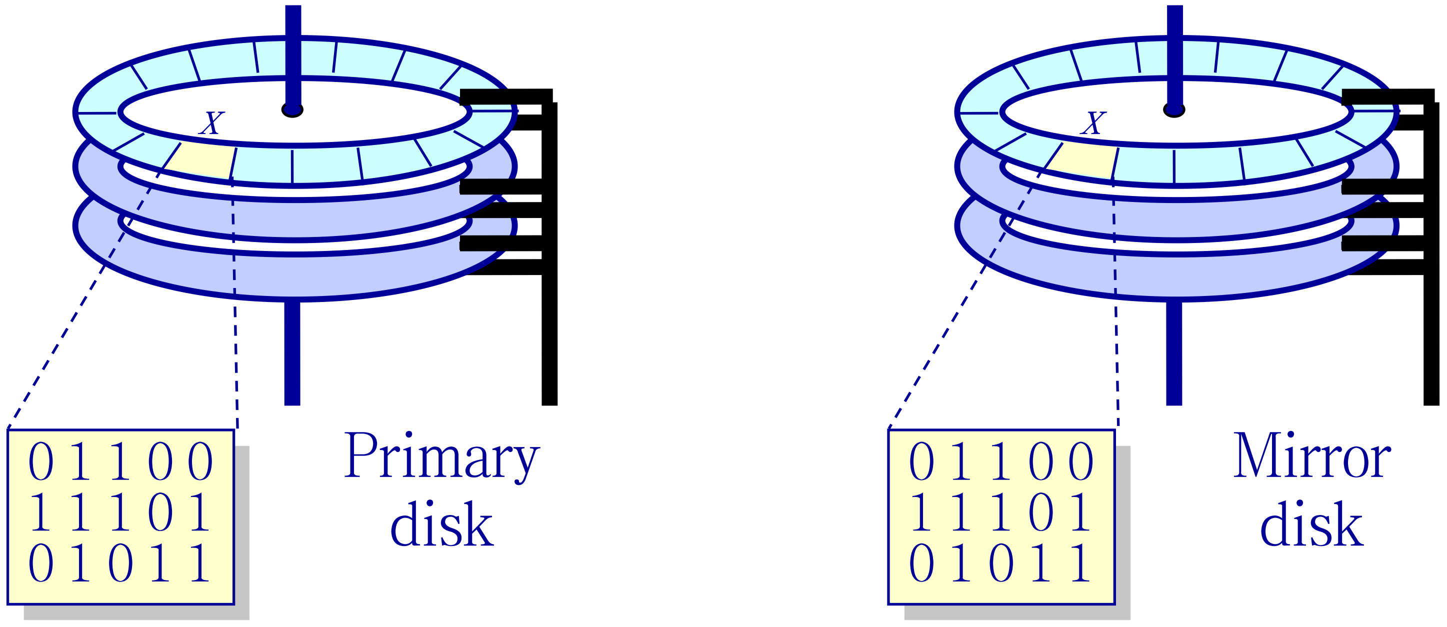
RAID-0: Disk Striping for Throughput

- Blocks broken into sub-blocks that are stored on separate disks
  - similar to memory interleaving
- Provides for higher disk bandwidth through a larger effective block size



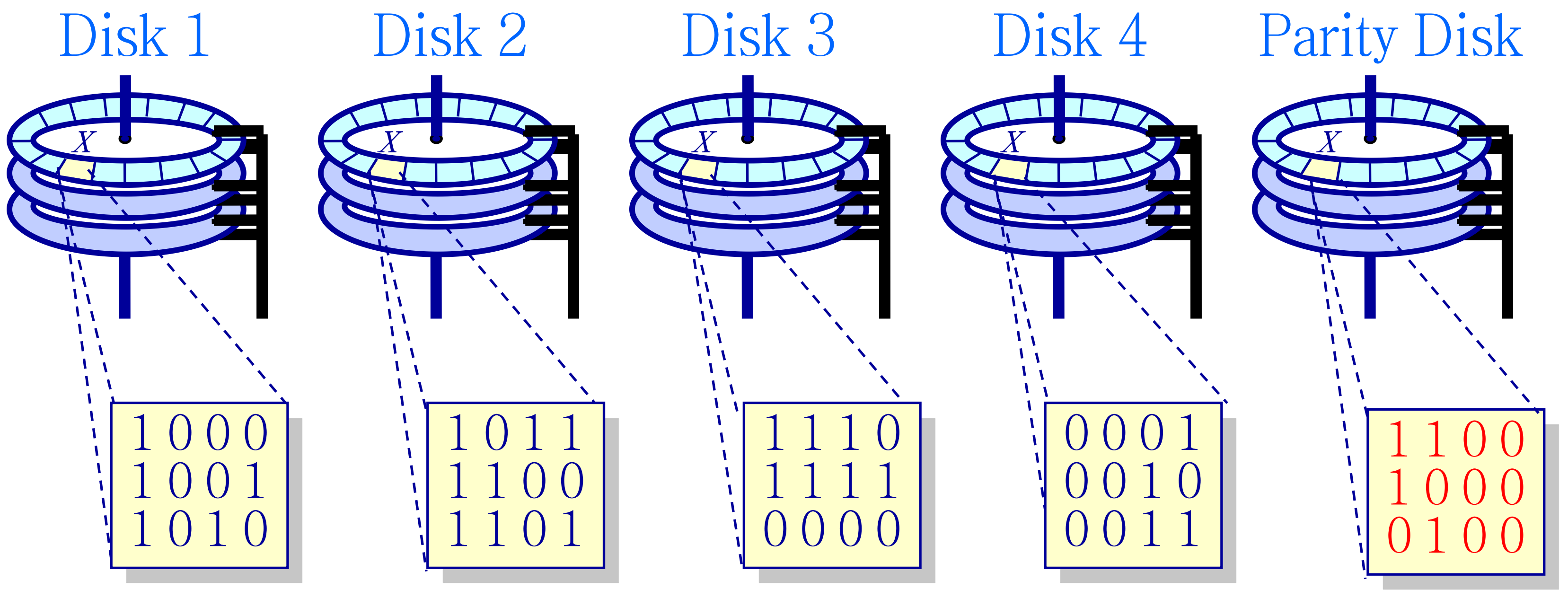
Raid-1: Disk Mirroring for Reliability

- Reliability is increased exponentially
- Read performance goes up linearly
  - Write to both disks, read from either.

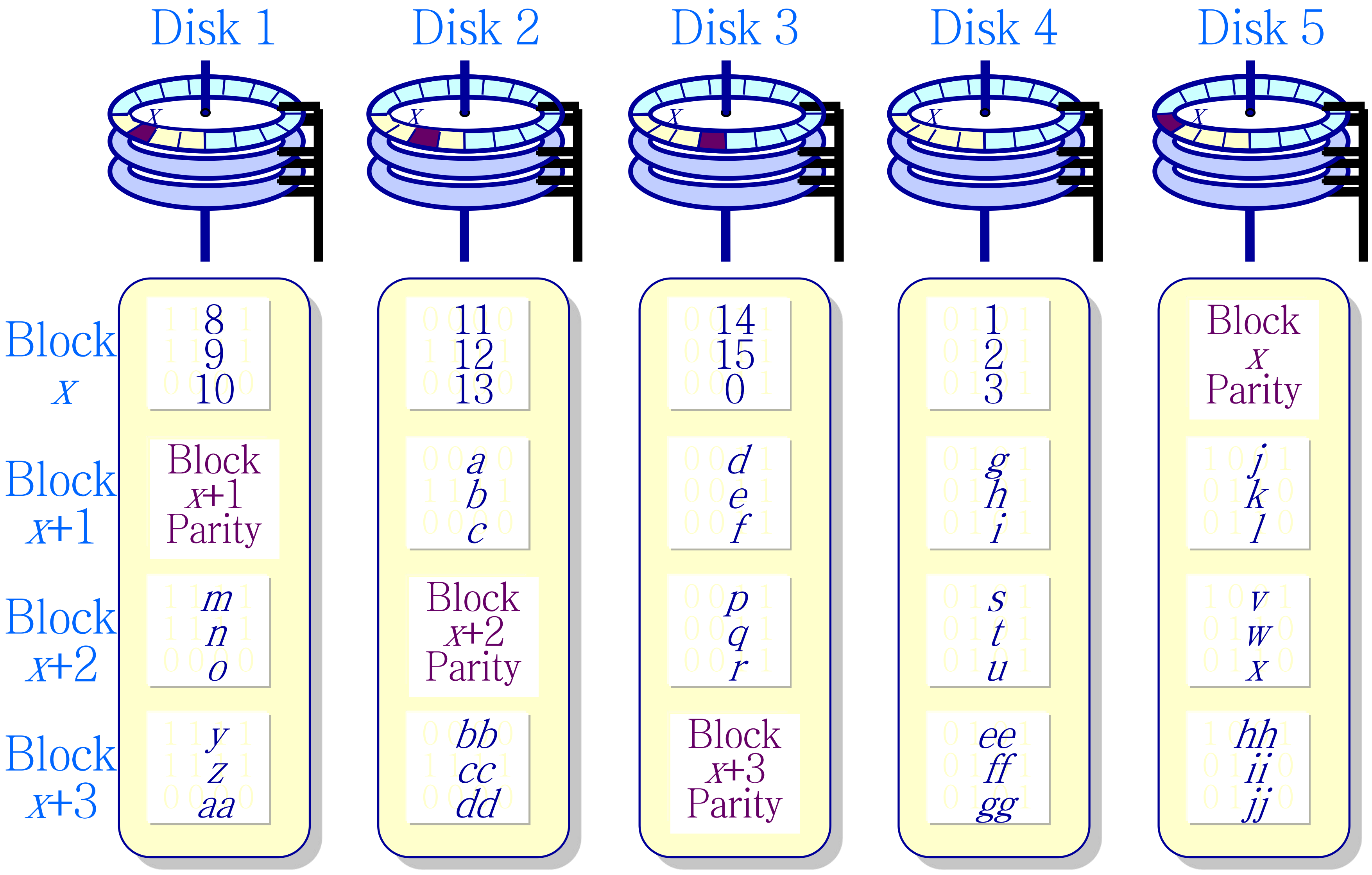


RAID-4: Parity Disk for Reliability

- Block-level striping with a dedicated parity disk
  - Allows one to recover from the crash of any one disk
  - Example: storing 8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3

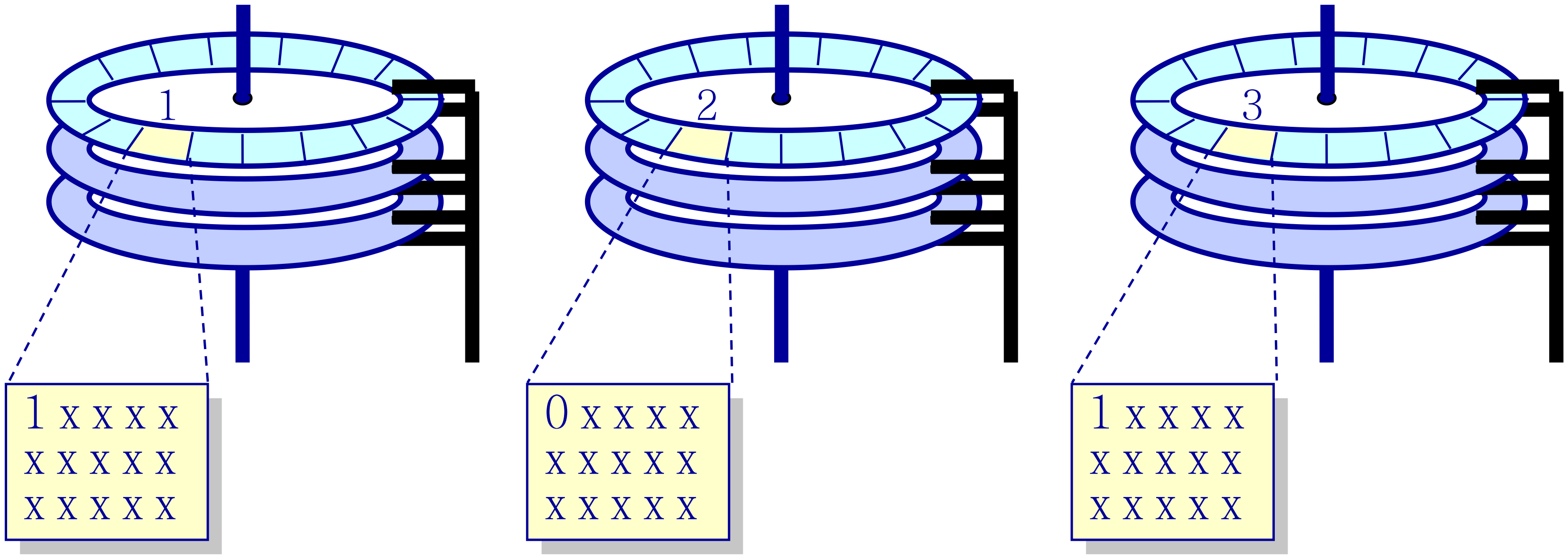


RAID-5: Block-interleaved Distributed Parity



Bit-wise vs Block-wise Disk Striping

- Striping and parity can be done byte-by-byte or bit-by-bit
  - RAID-0/4/5: block-wise
  - RAID-3: bit-wise
- Example: storing bit-string 101 in RAID-3 system

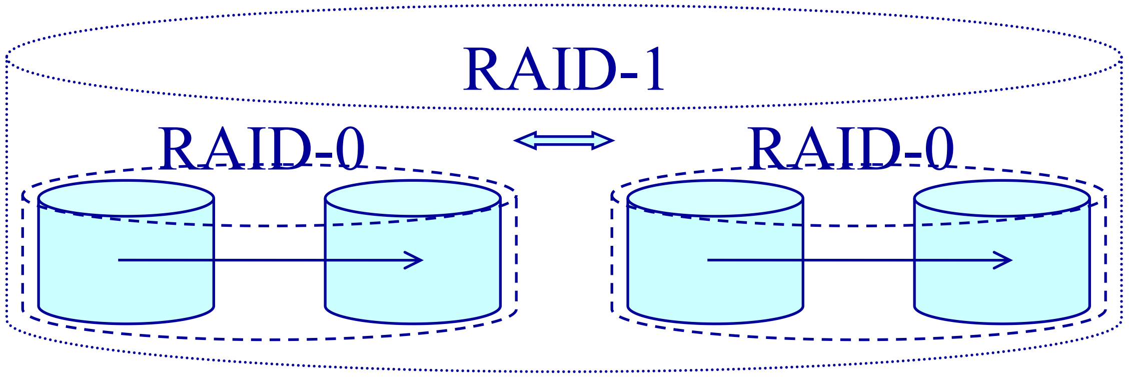


Tolerating Two Disk Failures

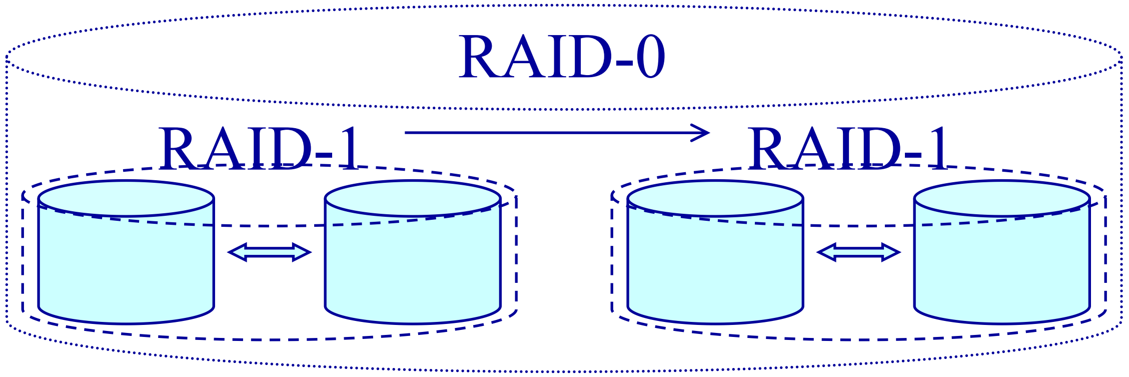
- RAID-5: single parity block per striping data block
  - tolerating one disk failure
- RAID-6: two redundancy blocks
  - With a special coding scheme
  - tolerating two disk failures

Nested RAID Levels

- RAID 0+1



- RAID 1+0


RAID-0
RAID-1
RAID-1