CMPT 300 — Operating Systems I

Summer 1999

Segment 9:

Disk and I/O Management

Melissa O'Neill

Secondary Storage Allocation

Let's think about disk storage

- Disk seen as large array of logical blocks (c.f., frames)
- Logical block is the smallest unit of transfer (c.f., page size)
- Logical blocks are mapped onto the sectors of the disk sequentially.
  - Block zero is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through
    - Sectors in track
    - Tracks in cylinder
    - Cylinders on disk
- Allocation of disk space to a file is rather like allocation of memory space to a process…
- (Directories, partitions etc. we'll leave until later)

Disk as One File

Like giving all memory to one process

- Entire disk used for one "file"
- File is seen as one big stream of bytes
- Usually fixed size, writes overwrite old data
- To read data at position *p*,

```
block  = p / BLOCK_SIZE;
offset = p % BLOCK_SIZE;
```

- Random access is fast
- Sequential access is fast
- Useful for special cases
  - Databases
  - Paging files
- Operating system still has work to do, preserving the illusion of a stream of bytes, rather than discrete blocks

Contiguous Allocation

More than one file per disk, all space per file is contiguous

- Each file occupies a set of contiguous blocks on the disk.
- Housekeeping is simple, for each file *f*

```
start_of(f)  — the block where f begins
length_of(f) — the block where f begins
```

- To read data at position *p*, in file *f*

```
block  = start_of(f) + p / BLOCK_SIZE;
offset = p % BLOCK_SIZE;
```

- Random access is fast
- Sequential access is fast
- External fragmentation when files are deleted
  - Compaction?
- Files cannot grow :-(
  - (Unless there is some empty space ahead of them)

Linked Allocation

Make each block in a file point to the next one

- Housekeeping is simple, for each file *f*,

  start_of(*f*) — the block where *f* begins

  and each block, *b*, has a pointer, of size PTR_SIZE, such that

  next_block(*b*)— the block that comes after this one

- To read data at position *p*, in file *f*

  ```
  block  = find_block(start_of(f),
                      p / (BLOCK_SIZE - PTR_SIZE));
  offset = p % BLOCK_SIZE;
  ```

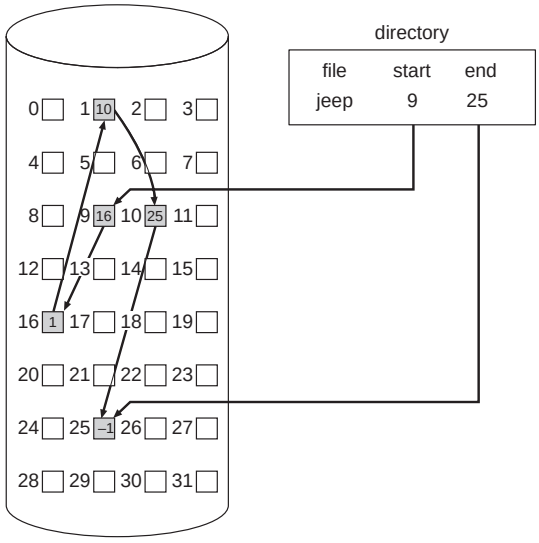
  where, find_block is defined as:

  find_block(*b*, 0) = *b*

  find_block(*b*, *n*) = find_block(next_block(*b*), *n* – 1)

- No external fragmentation (with suitable free space management)
- Random access is slow
- Sequential access is okay

Linked Allocation



Indexed Allocation

Bring all pointers together in an index block (c.f., page table)

- Housekeeping is simple, for each file *f*

  index_block(*f*) — the location of *f*'s index block

  and each index block, *b*, has a pointers to *f*'s blocks

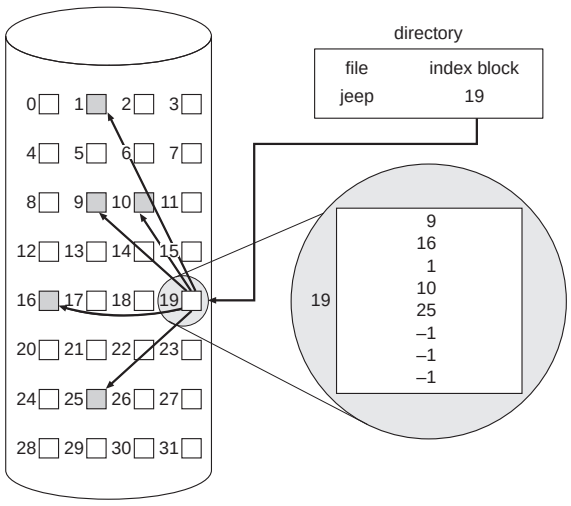
  index(*b*, *i*) — the *i*th entry in index block *b*

- To read data at position *p*, in file *f*

  ```
  block  = index(index_block(f), p / BLOCK_SIZE);
  offset = p % BLOCK_SIZE;
  ```

- Random access is okay
- Sequential access is okay
- Dynamic access without external fragmentation
  - But have overhead of index block (incl. fragmentation)
  - Limits on size of index block are limits on size of files (if we use two bytes for block number and 512-byte blocks, max file size is 128K)

Indexed Allocation (contd.)

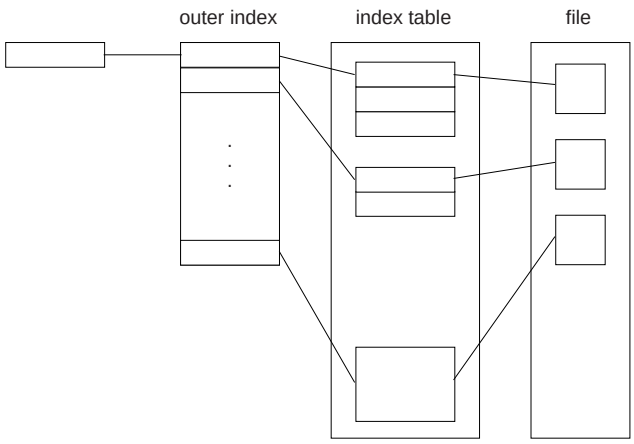


Indexed Allocation (contd.)

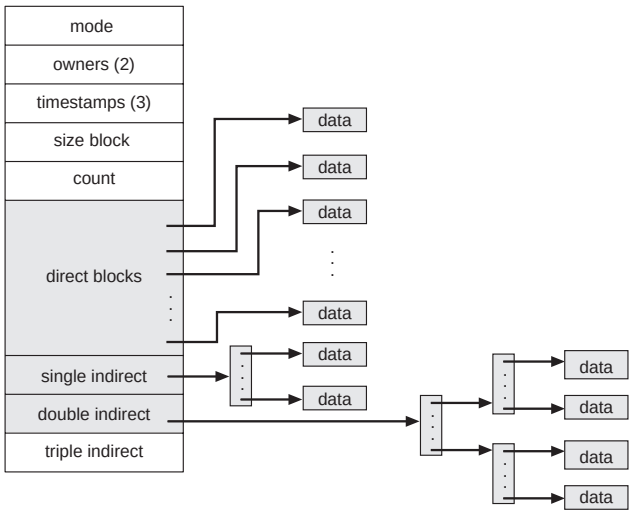
Handle larger files by:

- Increasing logical block size
  - But more internal fragmentation
- Using larger index blocks
  - But the index blocks are too much like files (different sizes, complex to manage).
- Using a two-level indexing scheme

Two-level Indexes



Combined Indexing



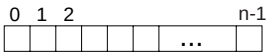
Performance of Index-based Allocation

Indexed allocation

- Allows a file's blocks to be scattered all over the disk
  - But scattering the blocks slows performance
- Try to locate the file (or large pieces of the file) in the same region of the disk
  - Minimize disk head movements required
  - Requires enough free space to be able to pick a region of the disk that has chunks of space free

Free-Space Management — Bit Vector

Bit map for n blocks…



bit[*i*] = { 0 ⇒ block[*i*] free
1 ⇒ block[*i*] occupied

- Easy to find free space (find first zero bit)
- Easy to find contiguous blocks of free space
- Space hungry — bitmap for an 8 GB disk with 512-byte blocks would be 2 MB
- Bitmap does not get smaller as free space decreases

Class Exercise: How would we make the bitmaps take less space? (And what would the percentage be?)

Logical Block Size vs. Physical Block Size

Our filesystem need not be tied to the block size of the actual device. We can make the "logical block" size bigger (sometimes called the *cluster size*).

- 4K logical blocks are a convenient size (especially if this size matches our page size).
- A 4K logical block is written in 8 contiguous sectors on the disk.
- Reduces bitmap from 2MB to 256KB (or 0.003% of the disk).
- We can make the logical blocksize larger, but the larger the logical block size, the more internal fragmentation.

Free-Space Management — Free List

Represent free space as a linked list (just like a file with linked allocation)

- Very easy to find free space
- Space efficient — blocks used to hold free list are spare, so no overhead
- Free space may be scattered — hard to find contiguous blocks

Class Exercise: How would you store the free list to avoid problems with file fragmentation?

Region-based Approaches

Divide disk into regions (sometimes called *cylinder groups*), each with its own free list.

- Unless a file is very large, try to keep all of it in the same zone
- Try to put all the files in a directory in the same zone
- Put different directories in different regions

Class Exercise: What assumptions are we making here? What kinds of locality are we expecting?

Log-based Approaches

Change our assumptions

- Contiguous allocation would be great if we could only make it work
- Temporal locality rather than spatial locality
- Lots of memory for buffering
- Computers (and their disk drives) spend much of their time idle, so we can employ some kind of compaction scheme that runs as a background process

Write the file in lumps, making the lumps as big as possible

- Delay disk writes as long as possible
- Always write to the end of the "log"
- New space for the log is made by copying garbage collection

Layers in Action — Filesystem

Low-level filesystem

- Files don't have names/directories, just numbers (sometimes called the *inode number*).
- Implementation is easy

For example, Unix's FFS statically allocates a fixed number of inodes, scattered across the disk (following the region strategy).

- The 1GB drive on my workstation has 157,696 inodes, of which 34% are used (the disk itself is 99% full).
- You can see the inodes associated with your files on a Unix system by typing `ls -i`, for example:

```
 96746 Bibliography/   64029 intro.tex   64031 main.tex
119386 CVS/            64036 main.aux    64001 melissa.sty
125417 Diagrams/       64049 main.bbl    64054 method.tex
120966 Graphs/         64038 main.blg    64030 performance.tex
 63933 appendix.tex    64037 main.dvi    63999 preface.tex
 63998 conclusion.tex  64035 main.log
```


Layers in Action — Filesystem (contd.)

High-level filesystem

- Provides mapping from files/directories to inode numbers
- Implementation is easy — the directories can be files in the lower-level filesystem.
  - In Unix, directories are just files containing variable-length records that include just the name and the inode number. All the other data is included in the file's index node (*inode*).

Class Exercise: If we store data (permissions, ownerships, etc.) in the inode, doesn't this violate the two-layer scheme?

Metadata — What to store about files…

Class Exercise: What information should the operating system store about files?

Creator — Who made the file

We might want to store

- The user
- Their role
- The program

(Macintosh stores the creator; Unix conflates ownership with creator.)

Ownership — Who the file belongs to

Unix stores two ownership attributes:

- User
- Group

Where groups are system-wide groups of users.

A different operating system might do things differently:

- List of users
- List of groups

Groups could be user-defined or system-wide.

Access Rights

A user might be allowed one or more of the following access rights to a file:

- Existence check
- Execute
- Read
- Append
- General update
- Change access rights
- Delete

Access — Who can access the file

Vanilla Unix provides access based on

- User
- Group
- World

where each can set their own protection.

Other options include:

- List of users
- List of groups
- List of programs
- Sensitivity labels

Watchdogs

Let files/directories declare a program as their guardian

- Maximum flexibility
- Slower performance

Access Information

When the file was

- Created
- Data Modified
- Meta-Data Modified
- Data Read
- Meta-Data Read

and by whom.

We might want to have just information for the last access, or we might want to keep a log of all accesses, perhaps with *rollback* information.

File Types

What kind of file it is

- Executable
- Internal format (object file, TIFF image, Rich Text)
- Logical Record Type
- File type for OS
  - Lockable
  - Has ACL or watchdog
  - "Sticky"
- File organization
  - Sequential
  - Indexed
  - Random

File Types (contd.)

Often file name and contents can supplement file types provided by the operating system, but

- Not always elegant
- Not always efficient

In the past, operating systems provided many different file types, and many different file organizations. But,

- Inflexible
- Complicated the operating system

A simpler scheme is possible, using a logical record of one byte but allowing direct access (aka random access), where

- A contiguous block of logical records can be read/written atomically
- Allow blocks of logical records can be locked

Other…

Various other information

- Version
- Dependencies
- Expected size
- Number of links

Again, we can avoid complicating the operating system with some of this information

- CVS (or RCS or SCCS) can provide version control
- make can manage dependencies

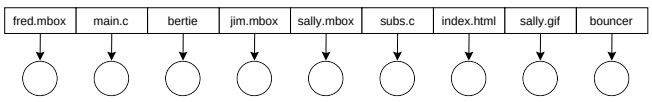
Directories — Motivation

- Convenience for users
  - Names allow user control, rather than machine control, of file identifiers
  - More efficient
- Logical grouping of files
- Many-to-one mapping (one file, many names)

Directories — Single Level

- Non-hierarchical
- Simple
- Inflexible
  - Naming problems
  - Grouping problems
- Inefficient search

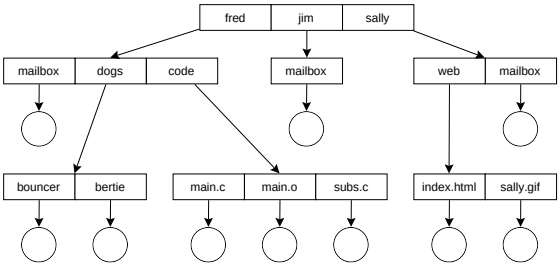
Example,



Directories — Tree Structured

- Tree-based hierarchy
- Betted Structured
- Efficient searching
- Grouping capability
- Directory Structures can be deep
  - Having to type a command like
    /usr/local/teTeX/bin/latex
    /Users/Melissa/Research/Papers/Determinacy/main.tex
    every time I want to typeset a paper I'm working on would be a pain
  - Introduce the notion of a *current directory* (and *relative paths*)
  - Introduce the notion of a *search path*
- Policy for directory deletion?

Directories — Tree Structured, Example

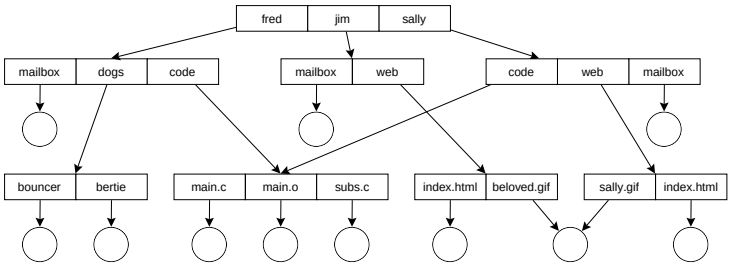


Directories — DAG Structured

- DAG-based hierarchy
- Allow sharing of files and directories (*aliasing*)
- Added complexity
- Implementation choices
  - Hard links (with reference counting)
  - Soft links (symbolic references)
- Policy for directory deletion?

Class Exercise: Allowing multiple hard links to a directory may be problematic — why?

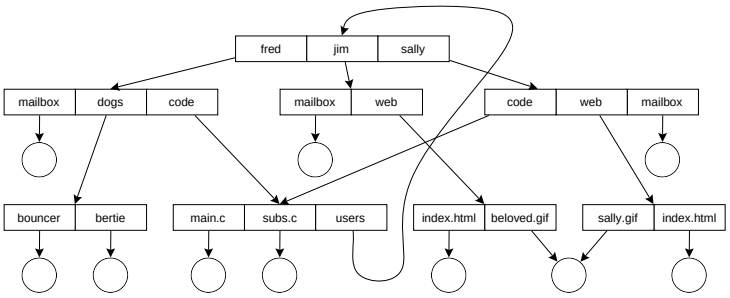
Directories — DAG Structured, Example



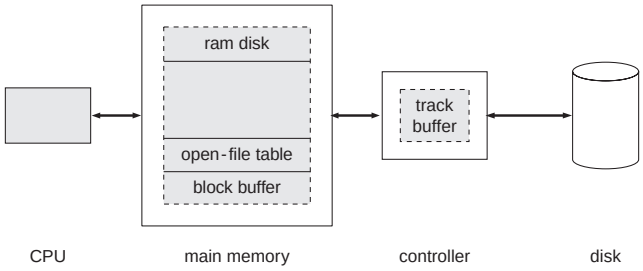
Directories — Graph Structured

- Graph-based hierarchy
- Most general
- Added complexity
  - Reference counting is inadequate for hard links to directories if cycles can form
  - Need garbage collection, or restrictive directory linking or deletion rules

Directories — Graph Structured, Example



Disk Caching



Class Exercise: If the operating system is going to cache blocks of a file in memory, what parts should be kept? (C.f., paging algorithms.)

Disk Caching (contd.)

Random access to file

- LRU replacement for cached blocks

Sequential access to file

- Free behind / read ahead

Disk Buffering

Allow writes to return immediately, after copying data into a buffer that will be written out to the disk

Buffers vs. Caches

- We *need* some kind of buffer (the write may not be the same size as the block)
- We don't need the cache, it just improves performance

Buffer-Cache

Both store disk blocks — use the same data structure, the *buffer cache*

- "Free" buffers used as cache
- Dirty buffers will eventually be written out to disk
- Allow buffer-cache to use any memory that is free in the machine.

Class Exercise: When should we write the dirty buffers?

Buffer-Cache (contd.)

One buffer cache for all processes and all block devices

- Local Disks
- Remote Disks
- Tapes, CD-ROMs, etc.

Dealing with System Failure

Class Exercise: Imagine a program is going to write to the end of a file (extending the file by one block).

In what order should we update the structures on disk to cause minimum damage if the system crashes?

(Assume the disk will never leave a disk block half-written.)

File creation?

File deletion?

Recovering from System Failure

Before system failure…

- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape).

After system failure…

- Run consistency checker — compares data in directory structure with data blocks on disk, and tries to fix inconsistencies.
- Recover lost file or disk by restoring data from backup.

Disk Scheduling

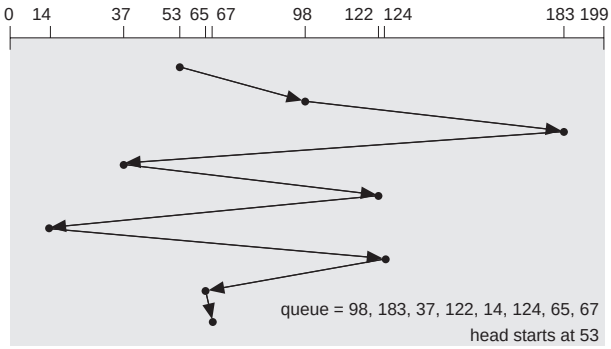
The operating system needs to use all I/O devices efficiently.

- Minimize access time — composed of
  - *Seek time* — time for the disk arm to move the heads to the cylinder containing the desired sector.
  - *Rotational latency* — additional time waiting for the disk to rotate the desired sector to the disk head.
- Maximize disk bandwidth
  - Bandwidth = Total Bytes Transferred / Total Time Taken
- Usually, disk requests can be re-ordered.
- Several algorithms exist to schedule disk I/O requests.
  - We illustrate them with a request queue of

    98, 183, 37, 122, 14, 124, 65, 67 (cylinders)

    and an initial head position of 53

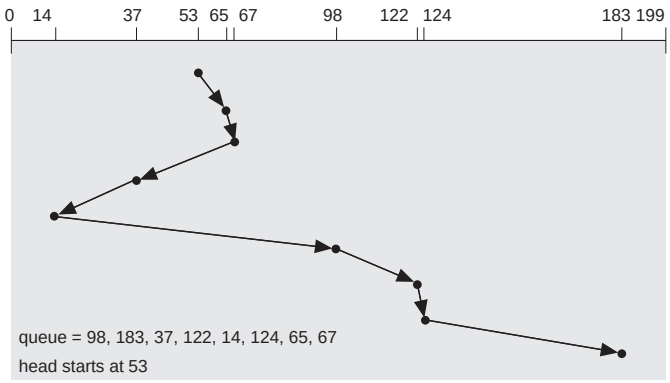
First-Come First-Served (FCFS)



Total head movement of 640 cylinders — Yuck!

Shortest Seek Time First (SSTF)

Select the request with the minimum seek time from the current head position.

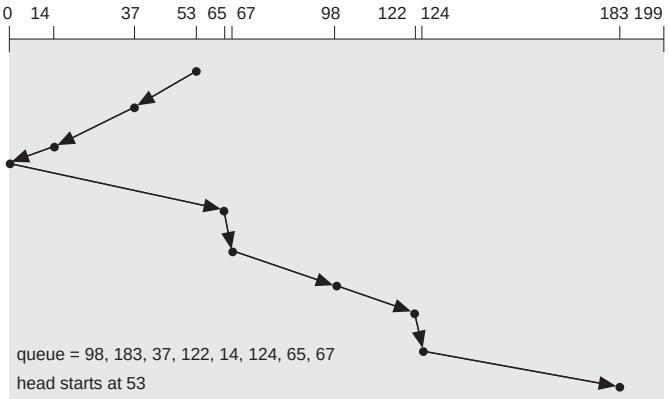


Total head movement of 236 cylinders

- May starve requests!

SCAN (aka The Elevator Algorithm)

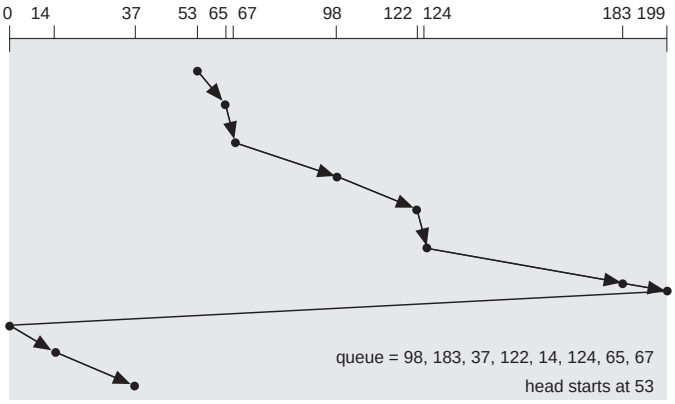
Move head from one end of the disk to the other, servicing requests as we go.



Total head movement of 208 cylinders

Circular-SCAN (C-SCAN)

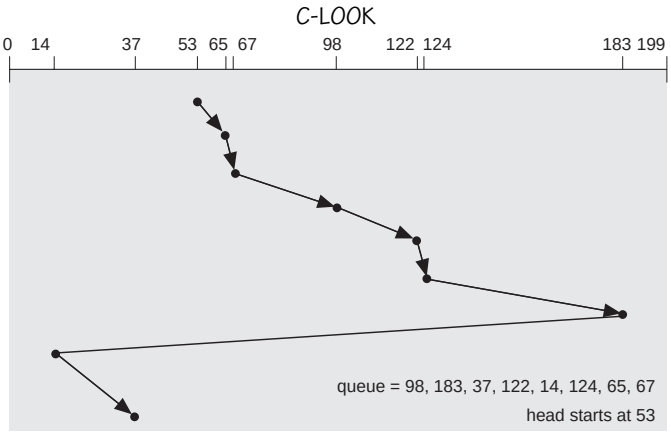
Move head from one end of the disk to the other, servicing requests as we go forward, zip back doing nothing.



Total head movement of 382 cylinders

LOOK & C-LOOK

Like SCAN and C-SCAN, but only go as far as the last request in each direction.



Total head movement of 322 cylinders (LOOK would be 180)

Selecting a Disk-Scheduling Algorithm

Comparing the algorithms, we find that

- FCFS is actually okay **if** the disk controller manages disk scheduling itself (many do these days)
- SSTF is common and has a natural appeal
- LOOK works well (lowest amount of head moment in our test)
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.

Modern Disk Geometry

Modern disks maximize disk utilization by putting varying numbers of sectors on different cylinders of the disk

- Logical block —> (sector, track, cylinder) complex or impossible for operating system to calculate.

Thus,

- Disk scheduling and scheduling grouping become impossible to perform perfectly.
- But approximations work.

More importantly,

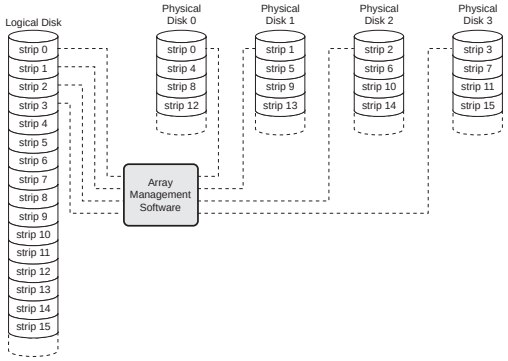
- Careful placement of vital data across platters may not work out.

Redundant Array of Independent Disks (RAID)

Disks are

- Less reliable than solid-state components
- Slow

RAID aims to address these issues by using multiple physical disks, operated in parallel to appear as one disk.



Redundant Array of Independent Disks (RAID)

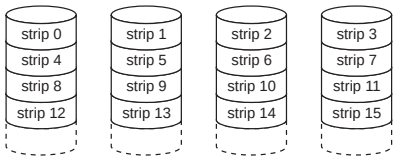
Strip size, choice of

- Physical disk block
- Filesystem logical block
- Track
- Cylinder
- …

RAID 0 (really _AID)

Data is striped across *n* disks

- No redundancy — single drive failure means data loss
- Maximizes parallelism
  - Every strip is stored on just one disk
  - Transfer rate up to *n* times the transfer rate of one disk, if *n* contiguous strips are accessed
  - Small strips (1/*n* of I/O request) => high transfer rate
  - Up to *n* strips may be accessed simultaneously
  - If strip size matches typical I/O request size, high I/O request rate

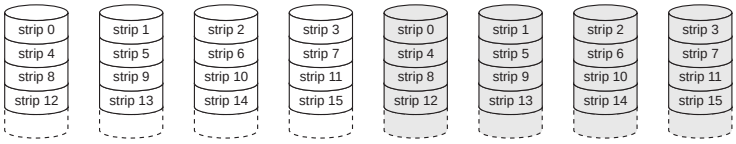


RAID 1 (Mirroring)

Duplicate everything, use 2*n* disks where RAID 0 would use *n*

- Worst case, data loss after two drive failures, best case, data loss after *n*+1 drive failures.
- Good parallelism
  - Every strip is stored on two disks (identical copies)
  - Read transfer rate up to 2*n* times the transfer rate of one disk, if 2*n* contiguous strips are read
  - Write transfer rate up to *n* times the transfer rate of one disk, if *n* contiguous strips are written
  - Small strips (1/*n* of I/O request) => high transfer rate
  - Up to *n* (or 2*n*) strips may be accessed simultaneously
  - If strip size matches typical I/O request size, high I/O request rate
- Most expensive
- Drives must detect errors themselves

RAID 1 (Mirroring, contd.)

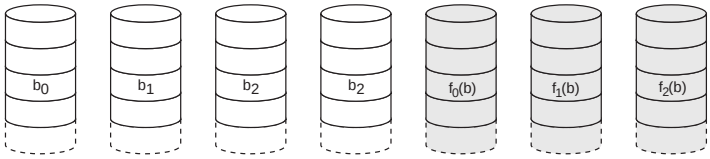


RAID 2 (Parallel Access with Hamming Code)

Use *n*+3 disks where RAID 0 would use *n*

- Data loss after two drive failures
- Fine-grained parallelism
  - Every strip is stored across all disks
    - Each of the *n* disks stores 1/n of the bits in the strip
    - The remaining three disks store hamming code
  - All disks accessed in lock-step
  - Read transfer rate up to *n* times the transfer rate of one disk
  - Write transfer rate up to *n* times the transfer rate of one disk
  - I/O request rate identical to that of a single disk
- Can detect/correct disk errors even if drive does not

RAID 2 (Parallel Access with Hamming Code)

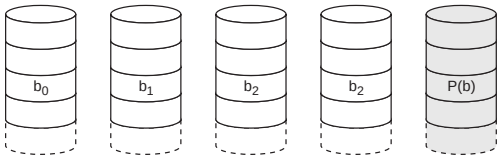


RAID 3 (Parallel Access with Parity)

Use *n*+1 disks where RAID 0 would use *n*

- Data loss after two drive failures
- Fine-grained parallelism
  - Every strip is stored across all disks
    - Each of the *n* disks stores 1/n of the bits in the strip
    - The remaining disks stores parity
  - All disks accessed in lock-step
  - Read transfer rate up to *n* times the transfer rate of one disk
  - Write transfer rate up to *n* times the transfer rate of one disk
  - I/O request rate identical to that of a single disk
- Drives must detect disk errors

RAID 3 (Parallel Access with Parity, contd.)



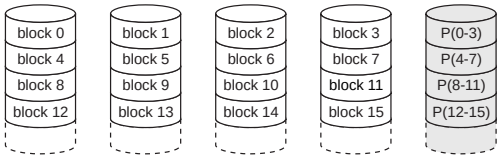
RAID 4 (Independent Access with Parity)

Use *n*+1 disks where RAID 0 would use *n*

- Data loss after two drive failures
- Coarse-grained parallelism
  - Every strip is stored across all two disks
    - Strips as in RAID 0 for *n* disks
    - The remaining disk stores parity
  - To write a strip, parity must be read, updated, and written, too
  - Read transfer rate up to *n* times the transfer rate of one disk, if *n* contiguous strips are read
  - Write transfer rate nearly *n* times the transfer rate of one disk, if *n* contiguous strips are written <u>at once</u>
  - Small strips (1/*n* of I/O request) => high read transfer rate
  - Up to *n* strips may be read simultaneously

RAID 4 (Independent Access with Parity, cond.)

- Coarse-grained parallelism (contd.)
  - If strip size matches typical I/O request size, high read request rate
- Drives must detect disk errors

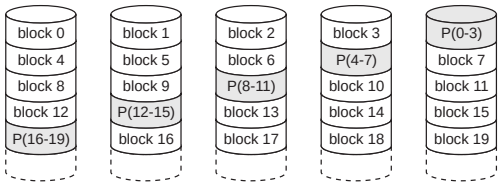


RAID 5
(Independent Access with Scattered Parity)

Use *n*+1 disks where RAID 0 would use *n*

- Data loss after two drive failures
- Coarse-grained parallelism
  - Like RAID 4, but we scatter the parity information across all the disks
  - Read transfer rate up to *n* times the transfer rate of one disk, if *n* contiguous strips are read
  - Write transfer nearly *n* times the transfer rate of one disk, if *n* contiguous strips are written <u>at once</u>
  - Small strips (1/*n* of I/O request) => high transfer rate
  - Up to *n* strips may be accessed simultaneously
  - If strip size matches typical I/O request size, high I/O request rate
- Drives must detect disk errors

RAID 5
(Independent Access with Scattered Parity)



(Open) File Tables

The operating system needs to know what parts of the file are in the buffer cache, and what parts aren't.

- Global information stored globally
- Some information can be per process

File Access

Programs need to deal with the data in files

- Read data in from files
  - Read whole file from start to finish
  - Read some data from the middle (or anywhere else)
- Write data to files
  - Append to the end
  - Change data in the middle

and manage the files themselves.

Class Exercise: What operations should we provide for accessing files?

Develop a stateless file interface and a stateful one.

Stateless File Access

No (apparent) internal state — each system call fully describes the desired operation

File access:

- read(filename, position, length, buffer)
- write(filename, position, length, buffer)
- truncate(filename, length)
- lock(filename, position, length, lock_state)

File management:

- status(anyname, info)
- delete(anyname)
- create_file(filename)
- create_dir(dirname)
- move(anyname, anyname)

Stateful File Access

Operating system keeps internal state

File access:

- handle = open(filename, mode)
- read(handle, length, buffer)
- write(handle, length, buffer)
- truncate(handle, length)
- seek(handle, position)
- lock(handle, length, state)
- close(handle)

File management:

- status(anyname, info)
- delete(anyname)
- change_directory(dirname)
- create_dir(dirname)
- move(anyname, anyname)

Contrasting Stateful vs. Stateless

Completeness

- Each method can simulate the other

Stateless operation

- Simple
- Works well in a multi-threaded program
- Basis for NFS

Stateful operation

- Assumes file-locality and sequential access is common
- Provides the operating system with more information about which files are being used
- Maps well to other kinds of device besides files (e.g., read/write to a terminal)
- May add arbitrary limitations (e.g, maximum open files)

Consistency Model

Additional complications

- Multiple processes can access files — Two (or more) processes could read and write the same file.
- Asynchronous writes + Errors = ?
- What if the file is moved/renamed/deleted while a process is using it?

What should the rules be?

Class Exercise: Develop and justify a consistency model for file operations.

Develop another one.

Unix Consistency Model

Unix uses the following rules

- All file operations are globally atomic
- A file is not removed from the filesystem until all links to the file are removed—an `open` counts as a link
- `write` in one process is globally visible immediately afterwards
- `write`s are asynchronous, I/O errors may not be discovered until the file is `close`d

These rules do not map well onto a stateless I/O interface

NFS Consistency Model

NFS uses the following rules

- All file operations are globally atomic and stateless
- Moved/rename/delete can disrupt accesses performed by other processes.
- write in one process is globally visible immediately afterwards
- writes are synchronous, I/O errors are discovered immediately

These rules map well onto a stateless I/O interface, but are not entirely consistent with the POSIX file model.

AFS Consistency Model

AFS uses the following rules

- All file operations are locally atomic
- A file is not removed from the filesystem until all links to the file are removed—an open counts as a link
- Changes to a file performed by another process will only be seen if the file is closed and reopened.
- writes are asynchronous, I/O errors may not be discovered until the file is closed

These rules map well onto a stateful interface, but are not entirely consistent with the POSIX file model.

File Access — Revisited

Besides system calls for protection there are other system calls we might want

- Find the capacity of the disk
- Eject the disk
- Discover whether any reads/writes have transparently failed.
- … *etc* …

We shouldn't have to extend the operating system's API every time we come up with something else to add.

Add generic method to talk to the lower-level layers

- ioctl(handle, request, buffer)

(some operating systems do all their I/O this way, the kernel just passes generic request messages from the application to the device).

File Access — Pseudo-files

An alternative to adding ioctl calls is to provide additional files/directories for programs to access.

For example, a Unix system mounting a Macintosh floppy must accommodate the Mac OS resource fork and finder information components that have no corresponding Unix counterpart.

- myfile.doc — the file's data fork
- .resource/myfile.doc — the file's resource fork
- .finderinfo/myfile.doc— the file's finder information

Where .resource and .finderinfo are both pseudo-directories.

File Access — Leveraging the VM System

Use the virtual memory system to provide file access

- Memory-map the file into memory
- Page faults retrieve the file from disk
- Copy-on-write or writeback semantics

Add the following system calls:

- map_file(handle, size, mode, address)
- unmap_file(address)

(This mechanism does not provide for extending or truncating the file.)

Tertiary Storage

Tertiary Storage is

- Low-cost media
- Usually removable media

Examples

- Floppy disk
- Zip
- Jaz
- CD-ROM
- CD-R
- CD-RW
- DVD
- DAT
- DLT

Tertiary Storage — Removable Disk

Usually given similar filesystem as that of secondary storage, except:

- Removable disks require a label
- Removable disks may contain a filesystem belonging to another operating system
- Removable disks may not be trustworthy
- Removable disks may be removed — impact for caching

Tertiary Storage — Tape

Filesystem does not map well onto tape, so applications are usually given access to the raw device.

- Typically, only one application can use the tape drive at a time
- Application decides on the data format for the tape
  - Portability issues
- The last write defines the current end of the tape.
- Random access is slow (potentially minutes!), if allowed at all
- But, with careful design, the same I/O system calls can be used for a tape drive that could be used as disk.

Jukeboxes

A removable-media drive can be made part of a jukebox

- Potentially huge capacity
- High latency (potentially minutes/hours to load a medium, if another process is accessing a different medium)

Reliability

Removable-media is often less reliable than fixed media

- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
- An optical cartridge is likely to be more reliable than a magnetic disk or tape.
- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the medium unharmed.

Cost

Cost should be considered carefully

- The cost per megabyte of hard disk storage is competitive with magnetic tape if only one tape is used per drive.
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years.
- Tertiary storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives.

Other I/O Devices

We can partition devices into two broad categories

- Block devices
  - Reads/writes fixed-size chunks
  - Allow random access

  Examples
  - Disk (fixed, removable, magnetic, optical, etc.)
  - Tape
- Stream devices
  - Reads/writes variable-size chunks (or 1-byte chunks)
  - Sequential access

  Examples
  - Network sockets
  - Terminal I/O
  - Mouse
  - Scanner

Other I/O Devices

We can partition devices into two broad categories

- Block devices
  - Reads/writes fixed-size chunks
  - Allow random access

  Examples
  - Disk (fixed, removable, magnetic, optical, etc.)
  - Tape
- Stream devices
  - Reads/writes variable-size chunks (or 1-byte chunks)
  - Sequential access

  Examples
  - Network sockets
  - Terminal I/O
  - Mouse
  - Scanner

Other I/O Devices — Graphics display

Specialized memory, I/O hardware handles frame buffer. Frame buffer is often *dual ported memory* to allow it to be read by the graphics system and written by the CPU at the same time.

Two options for OS management of graphics

- Allocate frame buffer memory to one (or more processes).
  - X-Windows
  - Display Postscript
- Have the operating system manage screen drawing
  - Microsoft Windows

Segment Review

You should be able to:

- Describe and contrast contiguous, linked and indexed allocation
- Describe strategies for managing free space and allocating blocks
- Divide a filesystem implementation into two layers
- Select metadata to store about files
- Implement file protection mechanisms
- Contrast directory structuring mechanisms
- Contrast buffering with caching
- Describe and select from several disk scheduling policies
- Describe and contrast different RAID storage strategies
- Design file access mechanisms
- Design and contrast consistency models for files
- Define and contrast primary, secondary and tertiary storage
- Categorize devices as block or stream devices