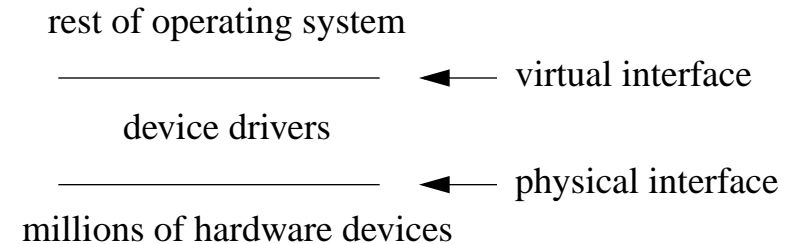# I/O and file systems

What abstractions does the OS provide above I/O devices (viz. disks)?

Device independence

Problem: lots of different brands of disks, disk interfaces, and disk controllers, each with their own custom control interface. Same is true of any I/O device (network card, video card, keyboard, mouse, etc.)

Add a "device driver" layer to hide this diversity/complexity

rest of operating system

—————————————— ◄——— virtual interface

device drivers

—————————————— ◄——— physical interface

millions of hardware devices

An abstraction makes things "better" for things that use the abstraction
  • threads: user doesn't have to worry about sharing CPU
  • addresses spaces: user doesn't have to worry about sharing physical memory (or physical memory size)
  • TCP: user doesn't have to worry about the unreliable network
  • device drivers: frees the user (the rest of the OS) from worrying about differences in devices (e.g. the device registers used to control the device)

Byte-oriented access -> block-oriented access

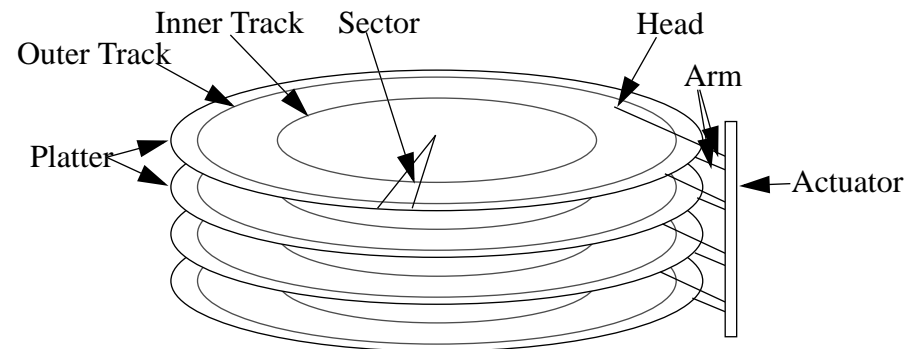Disks are accessed in terms of blocks (sectors), e.g. 512 bytes

But programs deal in bytes

How to read less than a block

How to write less than a block

Disk geometry and access

Disk geometry

- disk is made of a stack of spinning **platters**
- the top and bottom surface of each platter has concentric circles of data (called **tracks**). The set of tracks at the same radial distance is called a **cylinder**.
- each track is made of a number of **sectors**

Accessing a disk
- queueing time (wait for disk to be free): 0-infinity
- position disk arm and head (seek and rotate): 0-12 ms
- access disk data: size/transfer rate (e.g. disk transfer rate of 49 MB/s on Seagate Barracuda 1181677LW)

Optimizing disk performance

Disk is slow!

Best option is to eliminate the disk I/O (e.g. via file caching)

If you do have to go to disk, try to keep positioning time small

If you do have to re-position, try to get lots of data to amortize the positioning overhead

```
efficiency = transfer time /
             (positioning time + transfer time)

e.g. on Seagate Barracuda 1181677LW,
     transferring 300 KB takes 6 ms
     (50% efficiency)
```

Reducing positioning time

Optimize when writing data to disk
  • place items that will be accessed together near each other on disk

  • how to know in advance that two items will be accessed together?

Optimize when reading data from disk (minimize positioning time at time of access, rather than at time of creation)

Disk scheduling

Re-order a set of disk accesses to decrease the total positioning time
  • can be implemented in the OS, or in the disk hardware

FCFS (first come, first served)
  • e.g. (start at track 53): 98, 183, 37, 122, 14, 124, 65, 67
  • total head movement: 640 tracks (average 80 per seek)

SSTF (shortest seek time first)
  • e.g. 53->65->67->37->14->98->122->124->183
  • total head movement: 236 tracks (average 30 per seek)

  • any problems with SSTF?

SCAN: sweep the disk from one end to the other, then back again (sometimes called "Elevator algorithms")

  • e.g. 53->37->14->65->67->98->122->124->183
 • total head movement: 208 tracks (average 26 per seek)

Variants of SCAN
  • C-SCAN: always sweep the disk from beginning to end, then seek all the way back to the beginning and repeat.

  • LOOK, C-LOOK: when sweeping toward the end, stop if there are no requests beyond the current point.

Disk scheduling only improves positioning time with multiple outstanding requests (otherwise service the sole request)

File systems

A file system is an OS abstraction to make the disk easier to use

<div align="center">

user

——————————— ◄——— virtual interface

file system

——————————— ◄——— physical interface

disk

</div>
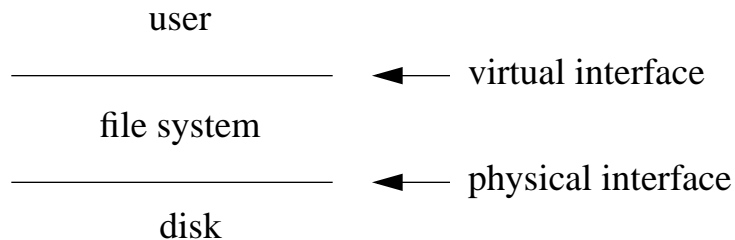
How to map file space onto disk space?
* file system structure on disk; disk allocation
* this will be similar to issues in memory management

How to use symbolic (english) names instead of disk sectors?
* naming; directories
* no equivalent issue in memory management, because the virtual and physical address spaces used the same type of "names" (i.e. addresses)

File system structure

Overall question: how to organize files on disk
* what data structure is the right one to use when storing a file on disk?

Need an initial structure that describes the object
* this is a "file header"; called an inode (index node) in Unix
* file header also contains miscellaneous information about the file, e.g. file size, modification date, permissions

Many ways to organize data on disk

Usage patterns guide our design choices
* 80% of file accesses are reads
* most programs that access a file sequentially access the whole file (the alternative is random access)

* most files are small; most bytes on disk are from large files

Contiguous allocation

Store a file in one contiguous segment on disk (sometimes called an "extent")

User must declare the size of the file in advance

Use some kind of fitting algorithm (e.g. first fit, best fit) to locate an area of disk that can store the file

File header is very simple: starting block #, file size

Equivalent to base & bounds for memory management

Pros and cons
    + fast sequential access (no seeks between consecutive disk blocks)

    + easy random access (can easily calculate the disk location of any file block)

    - external fragmentation

    - wastes space if user declares a bigger size

    - hard to grow files

Indexed files

User (or system) declares max # of blocks in file; system allocates a file header with an array of pointers big enough to point to that number of blocks

File header

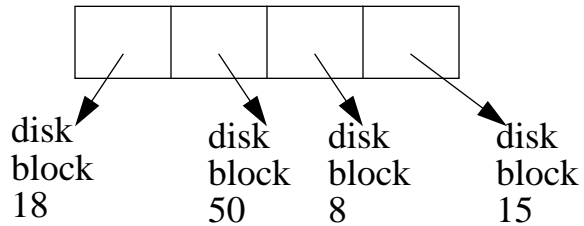| file block#	disk block #
0	18
1	50
2	8
3	15

Equivalent to a page table for memory management

Pros and cons
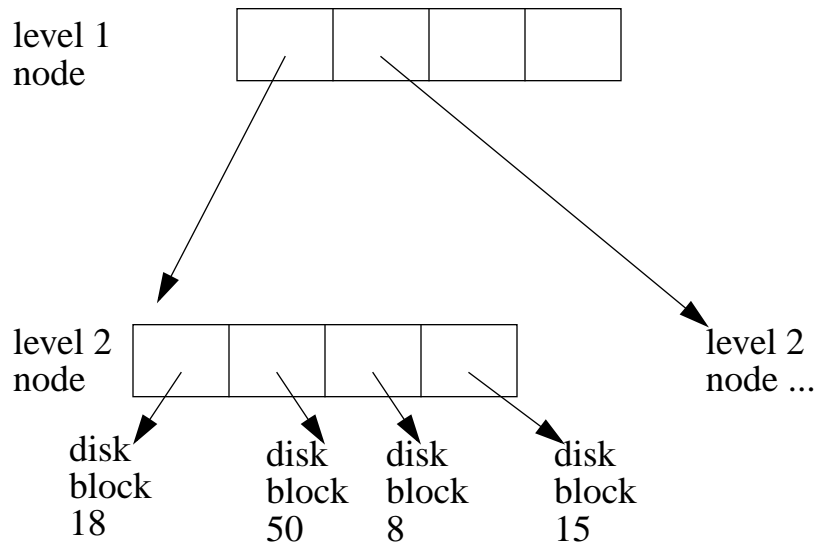   + can easily grow file up to the # of blocks allocated in file header
    + easy random access
    - how to grow file bigger than initially allocated in the file header?

    - lots of seeks for sequential access

Multi-level indexed files

Indexed files are like a shallow tree

```
┌────┬────┬────┬────┐
│    │    │    │    │
└────┴────┴────┴────┘
```

disk   disk   disk   disk
block   block   block   block
18   50   8   15

Could change this to a multi-level tree

level 1
node
```
┌────┬────┬────┬────┐
│    │    │    │    │
└────┴────┴────┴────┘
```

level 2
node
```
┌────┬────┬────┬────┐
│    │    │    │    │
└────┴────┴────┴────┘
```
level 2
node ...

disk   disk   disk   disk
block   block   block   block
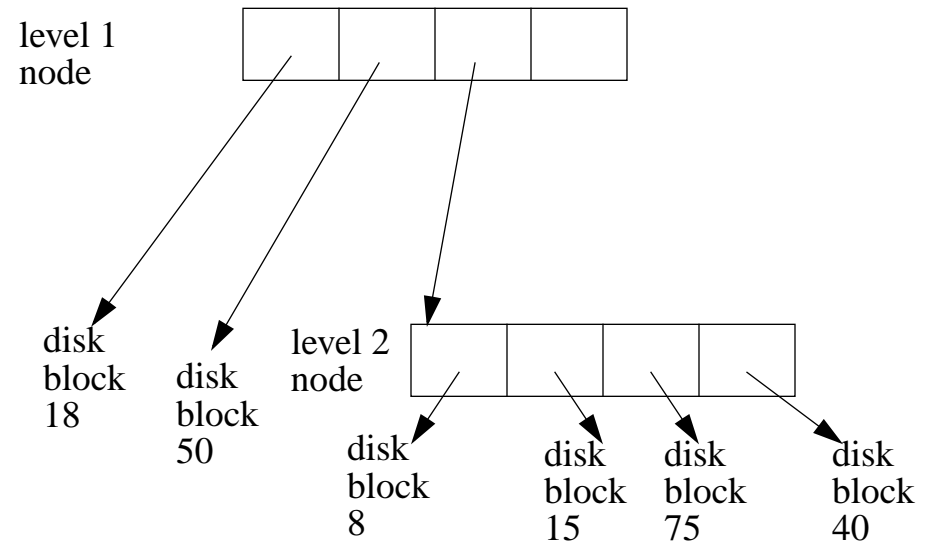18   50   8   15

This allows really big files without wasting header space for
    small files

How many accesses to get 1 block of data?

• how to solve?

Non-uniform depth multi-level indexed files

level 1
node
```
┌────┬────┬────┬────┐
│    │    │    │    │
└────┴────┴────┴────┘
```

disk   disk   level 2
block   block   node
18   50
```
                      ┌────┬────┬────┬────┐
                      │    │    │    │    │
                      └────┴────┴────┴────┘
```
             disk   disk   disk   disk
             block   block   block   block
             8   15   75   40

Pros and cons
  + files can easily expand
  + small files don't pay full overhead of deep trees
  - lots of indirect blocks for big files
  - lots of seeks for sequential access

How to reduce seeks for sequential access, and still avoid
  needing to pre-allocate a contiguous segment?

Naming

How to specify which file you want to access?
  • eventually, OS must find out which file header you want
    (viz. the disk location of the file header)

Typically, the user provides a symbolic name (e.g. english
  name, icon); OS translates name to a numeric file header
  (or disk location of the file header)
  • alternative is to describe the contents of the file, have the
    OS find the name by looking through the files for that
    content. This is usually done by databases on top of file
    systems.

Lots of ways to translate from symbolic name to file header's
  disk block #
  • e.g. hash table, expandable array, etc.
  • the data structure that does the mapping is called a **direc-
    tory**

Directories

A directory contains a mapping for all files in that directory
- name => file header's disk block # for that file
- often a simple array of (name, file header's disk block #) entries
- this table is stored in a normal file as normal data. E.g. "ls" can be implemented by reading this file and parsing its contents.

Directories are typically organized into hierarchical structure
- directory A has a mapping for a bunch of files **and directories** in directory A

E.g. /pmchen/482/names

/ is root directory
- contains a list of files and other directories
- for each file/directory in /, / has a mapping from name to file header's disk block #
- one of these entries is "pmchen"

pmchen is a directory entry within the / directory
- contains a list of files and directories
- one of the directories in /pmchen is 482

/pmchen/482 is a directory within the /pmchen directory
- contains a list of files and other directories
- one of the files it lists is "names"

How many disk I/Os to access the first byte of /pmchen/482/ names?

Current working directory: a shortcut for the user and the system
- cache the file header # for the "current" directory
- allows a user to specify a file name relative to the current directory
- allows the system to access a file without traversing the entire directory from /

Reducing positioning overhead by taking advantage of usage patterns

Combining (mounting) multiple disks into a single file system

Each disk has a file system
- each disk's file system starts at its own root (/)

Can tie several disks together into a single file system
- have an entry in one directory point to the root directory of **another disk's** file system (this entry is called a "mount point")

E.g. CAEN workstation

```
% ls /
afs/                    kernel/                 sbin/
bin@                    lib@                    template-server@
core                    lost+found/             template-server-ro@
dev/                    nfs/                    tmp/
devices/                opt/                    usr/
etc/                    platform/               var/
export/                 proc/                   vol/
home/                   root/

/bin is a normal directory in /
/usr points to the root file system of disk 1
/var points to the root file system of disk 2
/afs points to the root file system of a
      distributed file system
```

Windows: disks are made visible under my_computer/ {C,D,E}

Unix: can mount disks at any place in the directory hierarchy

File caching

File systems have lots of data structures on disk
- data blocks
- metadata: bitmap of free blocks, directories, file headers, indirect blocks, etc.

File caches speed access to all these types of data

Changing random disk I/O to sequential I/O can help somewhat
- throughput increases dramatically
- response time for small requests doesn't improve

Changing disk I/O to memory access helps a lot more
- throughput is better even than sequential disk I/O
- response time can improve by 100,000x

Should file cache be stored in virtual or physical memory?

Comparing file caching and virtual memory

Both use physical memory as a cache for disk
- VM: started with memory and added disk to get larger memory
- file systems: started with disk and added memory to get faster performance

Why have two mechanisms that both cache disk data in memory?

Memory-mapped files
- use the VM paging system to cache both virtual address spaces **and** files
- map a file into a virtual address space, then mark the backing store for the file as being allocated to the disk blocks for that file
- VM knows how to cache address spaces; file cache knows how to cache files. Memory-mapping makes files look like an address space, so VM cache can deal with it.
- e.g. how to load a program from disk into memory?

Caching issues

Normal design dimensions that you encounter with all caches, e.g. cache size, block size, replacement, etc.

Should the file cache use write-through or write-back?

Multiple updates and reliability

Reliability (durability) is especially important to file systems
- data in a process address space need not survive a system crash
- user considers data in a file system as permanent

Multi-step updates cause problems if crash happens in the middle

E.g. transfer $100 from my account to Janet's account
```
1. deduct $100 from my account
2. add $100 to Janet's account
```

E.g. move file from 1 directory to another
```
1. delete file from old directory
2. add file to new directory
```

E.g. create (empty) new file
```
1. update the directory to point to the
   new file header
2. write new file header to disk
```

What happens if you crash between step 1 and 2?

Careful ordering

How to fix the problem in the prior example?

Can careful ordering solve the problem of transferring money from Peter's account to Janet's account?

What 482 concept does this remind you of?

Transactions

Used a lot in databases. For us, the main component is atomicity (all or nothing)

```
begin
    disk write
    disk write
    disk write ...
end (this is "commits" the transaction)
```

Basic atomic unit provided by the hardware is writing a single sector to disk

How to make an arbitrary sequence of updates atomic using a single sector update?

Shadowing
- keep two versions of the database (old and new), and keep a pointer to which is the current version
- update the new version, then switch the pointer to the new version when you want to commit the changes

Logging
- write new data to a append-only log
- write "commit" sector to the end of the log after all the changes
- copy the new data to the original copy

- what if the system crashes after writing the log record (including commit) but before writing the in-place copy?

Almost all file systems since 1985 use transactions (using logging) to make atomic a group of updates to the file system metadata

- why don't file systems provide atomic transactions for data updates?

Distributed file systems

Distributed file system: file system that distributes where data is actually stored, but unifies the view of this distributed storage.
- allows access to files stored on remote computers (with same interface as to local files).

Benefits
- share data and resources between people
- enables uniform view from different machines

Basic implementation
- ask appropriate server for file (classic client/server model)
- but poor performance. How to improve?

Caching in a distributed file system

E.g. I am on azure and want to access the file /afs/engin.umich.edu/class/perm/eecs482/bin/submit482, which is on bighouse.engin.umich.edu

What happens to the file?

Transfer the sole copy from the server to my client? Does this reduce network traffic?

Make a copy on my client (replication)
  • what if I modify my copy?

  • when do I have to give up the copy (called invalidation)

State of a client's copy of a file