Computer Science 322
Operating Systems
Mount Holyoke College
Spring 2008

Topic Notes: File Systems

Disks and Disk Structures

We will consider disks and file structures in much more detail than most of the other types of I/O devices.

Data is written to the surface of the disk. How can it be arranged?

CD/DVD is arranged in a "spiral" for a continuous stream.

We'll concentrate on magnetic disks (floppy disk, hard disk). A hard disk may have multiple surfaces, or platters. For simplicity, assume there is only one disk, or platter, involved.

A read/write head is needed for each platter.

The data on a disk is arranged in concentric rings called *cylinders* or *tracks*.

Each cylinder of the disk is divided into chunks called *sectors* that contain *blocks*, the minimum allocatable and addressable unit on the disk. Since there is more space on the outside of the disk, there may be more blocks in outer cylinders than there are on inner cylinders.

The particular configuration of cylinders, sectors and the number of platters is the *drive geometry*. The actual drive geometry may be difficult to determine, as modern disk drives lie, controllers lie, and by the time you get the numbers they may be completely meaningless.

So to read or write data on the disk, a cylinder and sector must be specified. The read/write head must be positioned over the desired cylinder and sector. The read/write heads are typically connected to the end of a moveable arm. This arm is moved to position the head at the correct cylinder. When the disk rotates and the desired sector reaches the read-write head, the read or write operation can proceed.

The speed of this operation depends on two major factors:

- *seek time* – the time it takes to move the read/write head to the correct cylinder
- *rotational latency* – the time it takes for the correct sector to rotate under the read/write head

We can minimize seek time by minimizing the distance the read/write head has to move in order to service the incoming requests.

File System Interface

We switch focus now to talk about how to organize information on disks.

Hopefully everyone has a good idea what we mean by a *file*.

- files can be data or programs
- can be simple or complex (plain text, or a specially-formatted file)
- structure of a file is determined by both the OS and the program that creates it
- files are stored in a *file system*, which may exist
  - on a disk
  - on a tape
  - in main memory

Files have a number of attributes:

- filename
- file type (maybe)
- location – where is it on the device
- size
- protection/permissions (maybe)
- timestamp, ownership
- directory information

A number of operations can be performed on files, many of which we have been doing without giving it miuch thought:

- create
- write/append
- read
- seek (reposition within file)
- delete
- truncate
- open, close

File Types

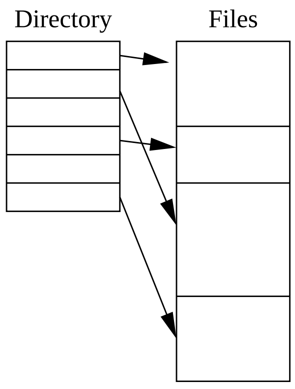
How does a “type” get assigned to a file?

- can use file extension (.c, .exe, .doc, .tex, .mp3, etc.)
- file extension may or may not be functionally important – even if not, they give the OS and the user an hint as to the type of the file
  - Windows uses an unenforced file extension registration
  - Macintosh can enforce types within a file – special part of a file called a “resource fork” to store extra information including the application that created it
  - Unix uses “magic numbers”. See the `file` command and `/usr/shared/misc/magic` in FreeBSD, `/etc/magic` in Solaris.

File access may be *sequential* or *direct*. Tapes support only sequential access, disk files may support both.

Directories

A listing of the files on a disk is a *directory*.



Both the directory structure and the files reside on the disk.

The directory may store some or all of the file attributes we discussed.

A directory should be able to support a number of common operations:

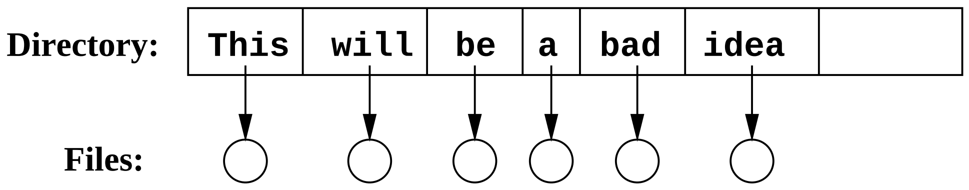
- search for a file
- create a file
- delete a file

- rename a file
- listing of files
- filesystem traversal (`cd`)

There are many ways to organize a directory, with different levels of complexity, flexibility, and efficiency. We will look at several possibilities.

- Single-Level Directory

  The simplest method is to have one big list of all files on a disk.



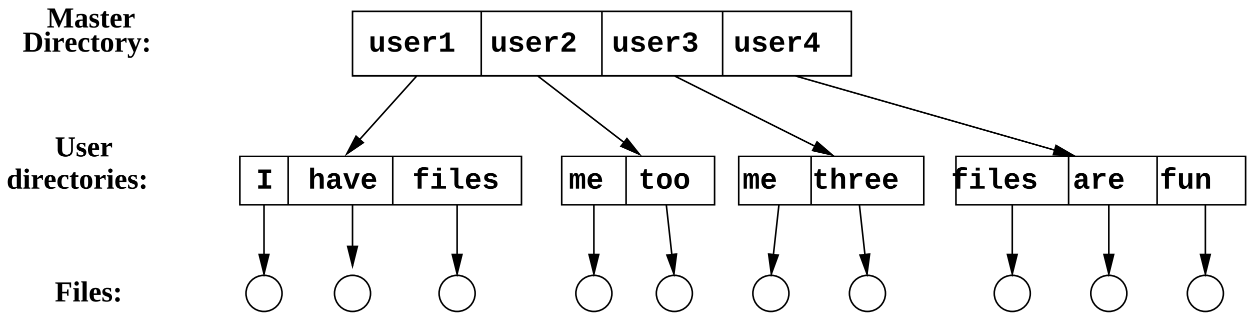
  This can be used for a simple system. A disk for the C-64 worked like this.

  But, it breaks down pretty quickly:

  – cannot have two files with the same name – could be necessary for multiple users/programs on a disk
  – no way to group files – just one big list
  – searches need to look through the entire directory

- Two-Level Directory

  We can create a separate directory for each user:

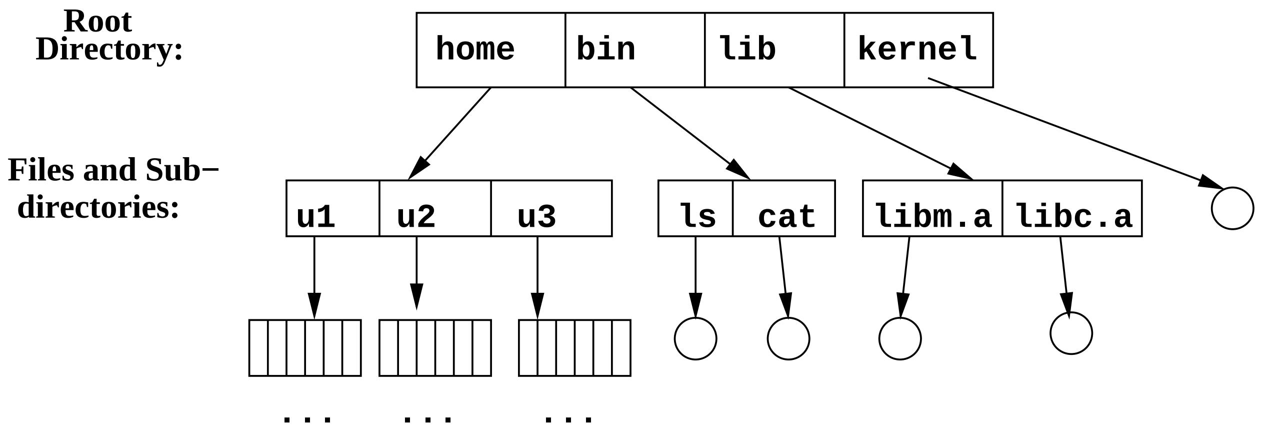


  – files now have a path name `/user1/have`
  – different users can have the same file name (`/user2/me` and `/user3/me`)
  – searching is more efficient, as only one user's list needs to be searched

  - but still no grouping capability for a user's files

- Tree-structured Directory

  Something more reasonable and useable:

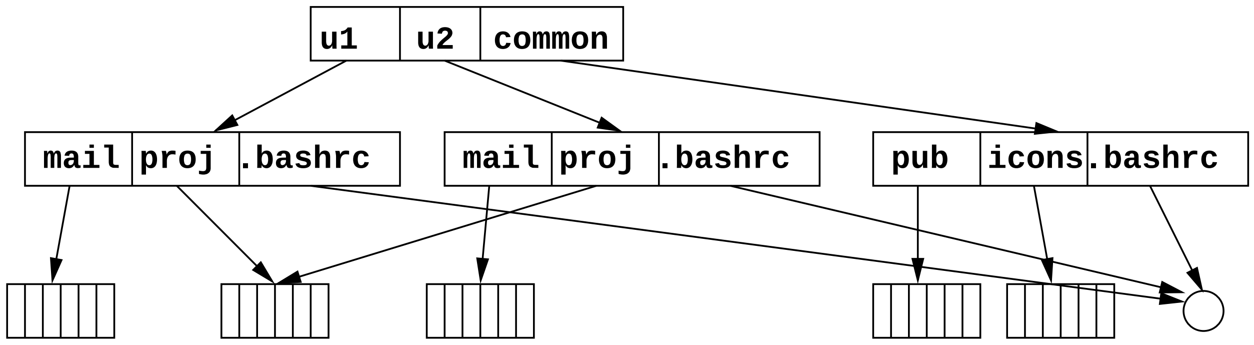


  - any directory entry can be either a file or a *subdirectory*
  - files can be grouped appropriately
  - search is more efficient – follow the path
  - add concept of a *current working directory*
    * traverse directory (`cd`)
    * operate on files in current directory by default
    * or specify path, either *relative* or *absolute*
  - need to be able to create and remove directories as well as files
  - consider: what happens when a non-empty directory is deleted?

- Acyclic-Graph Directories

  The tree model does not allow the same file to exist in more than one directory. We can provide this by making the directory an acyclic graph.

  Two or more directory entries can point to the same subdirectory or file, but (for now) we restrict it to disallow any directory entry pointing "back up" the directory structure.



These kinds of directory graphs can be made using *links* in the Unix world, *shortcuts* in the Windows world, or *aliases* in the Mac world.

We have multiple names for and multiple paths to the same file.

Unix links can be

- *symbolic* or *soft link* – specify a path to the file (logical) – `ln -s` – original file is "real" others are just pointing to that one
- *hard link* – actual link to the same file on the disk from multiple directories (physical) – `ln` – all hard links are equal

This allows sharing of files, but introduces complications – what happens when the file is removed from one of the directories? If there may be more references to the file, can we delete it? With symbolic links, the file just gets deleted and we have a *dangling pointer*. With hard links, a reference count is maintained, and the actual file is only deleted when all references to it are removed.

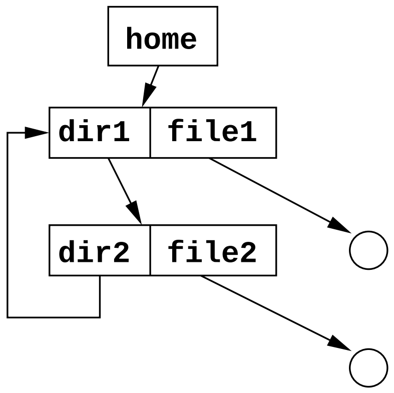
Demo: links. make directory with file. Create symlink, hard link. Look at `ls -l` output. Note special type of symlink, reference count of hard linked files. Also note that inode is the same for hard linked files (`ls -i` output). Demonstrate a dangling pointer. Demonstrate refcounts with copies. Demonstrate that a hard link cannot cross filesystem boundaries.

- General Graph Directory

  What if we allow links back up the chain?

  Unix directories have this built in – all directories except the system's root directory have a special entry `..` that indicates the parent directory, and an entry `.` that indicates the current directory.

  But they also allow links to be created back "up the chain" of the directory structures, potentially introducing cycles in the directory graph.



Demo: dirs is the above example in Unix. See how tcsh with its default behavior works. With `set symlinks=chase` it notices and truncates the path. The default behavior is from `symlinks=ignore`.

The long path is because my `pwd` is aliased to `echo $cwd`. `/bin/pwd` is not fooled. Note that the functioning of `cd ..` is a shell function here, as the actual current working directory does not include the cycles.

When general graph directories are allowed, we need to be careful with command like `find` that search a directory and its subdirectories for something. The search is infinite if cycles are followed. Typically, a program like `find` will not follow symlinks.

Problematic cycles can be avoided by allowing "up" links to files, not directories. Could also run cycle detection every time a new link is added, if this is a concern. Unix leaves it up to programs to make sure they treat symlinks appropriately.

BSD 4.3 limits the number of links allowed to be traversed for any given path name to 8 to avoid undetected cycles. This limit is actually 32 in the current version of FreeBSD, and 20 for Solaris.

See Example:
`/cluster/examples/morelinks`

Directory Implementation

In any case, an individual subdirectory will typically contain a list of files. How to store this list?

- Linear list – list of names, each of which has a pointer to the file's data blocks. This is straightforward, but requires a costly search on large directories.
- Hash Table – hashed linear list – decrease search time, but more complex to implement.

Another consideration: case sensitivity of filenames. Recent Windows, MacOS filesystems have filenames that remember case, but searches are case insensitive. Most Unix filesystems are truly case sensitive.

Disks and Partitions

A system may have a number of disks, each with one or more *partitions*. These are logical subdivisions of the physical disk, often created to help better organize data on the disk.

Demo: look at `df -kl` on mhccluster, also `fdisk`.

A partition is where a filesystem gets created (more on that soon). Once we have a filesystem, we need to make it accessible to the world.

In DOS/Windows, this typically involves assigning a letter to each partition. Then there is a directory hierarchy within each partition.

In Unix, there are no drive letters, everything is considered to be part of one big hierarchy. One partition forms the root directory (`/`) and all others are *mounted* into the structure it defines.

The places where partitions get mounted are called *mount points*, and are nothing more than regular directories. When a partition is mounted onto a mount point, the directory is replaced by the contents of the partition mounted.

When the mount point directory is accessed, the *virtual file system* layer of the OS notices that the directory has been used as a mount point, and sets the current directory to the root of the partition mounted there.

The partitions mounted can be of any type, but all appear to be part of the same directory structure. The system delivers requests to the appropriate partition, and a filesystem-type-specific set of operations are used to access the actual filesystem.

The list of partitions and their mount points and types are listed in a *file system table* file. In FreeBSD, it is located in `/etc/fstab`; in Solaris, it is `/etc/vfstab`.

The partitions mounted may be remote as well as local – more on this soon.

Disk Partitioning

Why might we partition a disk?

- logical separation of types of files (bootable OS, system programs, home directory space, shared space, scratch space) for security or backup purposes.
- want to run multiple OSs on the same system.
- separate partition to use as virtual memory (“swap partition”).
- to get around OS limits on the size of a filesystem when a single disk is larger than that limit.

How can we define these partitions? These are usually specified with a system disk management utility.

- DOS/Windows fdisk
- FreeBSD disklabel/bsdlabel
- MacOS Disk Utility

All of these do the same basic things. Break up the disk, usually on cylinder boundaries, into logical subunits.

Each of the partitions gets a device name, and in each of these we create a filesystem.

The filesystem can then be mounted at a given mount point (in the Unix world) or at a drive letter in DOS/Windows.

File System Implementation

Suppose we have partitioned a disk, and it's time to take those disk blocks that have been reserved for our partition and create an actual *file system* to hold our files.

The OS could just provide access to the blocks and let programmers deal with everything, but that's not very nice.

We want to provide those things that operating systems are supposed to provide:

- convenience
- protection
- efficiency

Several issues need to be considered:

- how do we allocate disk blocks within our partitions to files and directories?
- how we decide what blocks are available?
- what is the complexity and efficiency of the choices we make for those?

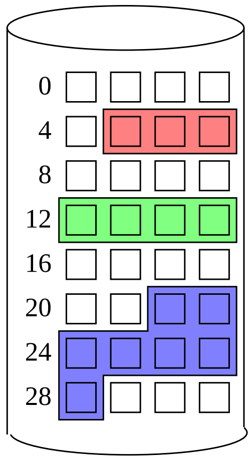
We have already talked about possible directory structures. Most likely, these directory structures will be implemented as files at some level. We'll need to be able to find them on the disk just like other files.

Allocation Methods

So first, we consider how disk blocks are allocated to files.

- Contiguous Allocation

  Each file is allocated a set of contiguous disk blocks



Directory:

file	start	length
`moo`	5	3
`snow`	22	7
`fall`	12	4

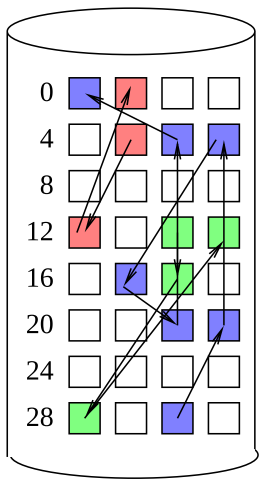
- similar to contiguous allocation of memory
- simple – directory entry needs only starting location (block number) and length (number of blocks)
- supports random access into files – can easily compute and read the block that contains a certain part of the file.
- can lead to holes (external fragmentation)
- may be difficult to have a file grow
- reading should be very efficient, since consecutive blocks of the file can be stored in consecutive blocks on the disk

• Extents

Extents are analogous to segmentation for memory allocation. Files are allocated as a collection of *extents*, which are contiguous chunks of disk blocks. Each has a starting block and a size.

• Linked Allocation

Each disk block has a pointer to the next disk block in the file as well as some file data.

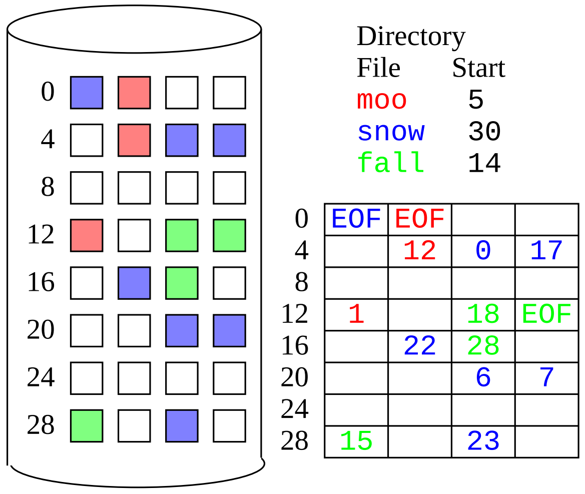


Directory

File	Start	End
moo	5	1
snow	30	0
fall	14	15

- need to reserve part of each data block for a pointer – can make for odd-sized data blocks
- directory entry requires only starting block
- easy to append to a file
- no external fragmentation
- no random access – have to traverse each block
- a bad disk block means the entire file from that block on is lost

A variation on this is the *File Allocation Table (FAT)* used by MS-DOS and pre-NT Windows versions. This gathers the links into one table.



Directory

File	Start
moo	5
snow	30
fall	14

0	EOF	EOF		
4		12	0	17
8				
12	1		18	EOF
16		22	28	
20			6	7
24				
28	15		23	

– get to use the whole disk block for data

– a bad disk block means only that block is lost

– unless... the FAT itself goes bad, in which case we have a problem – have backup copies on the disk, then run your favorite rescue program

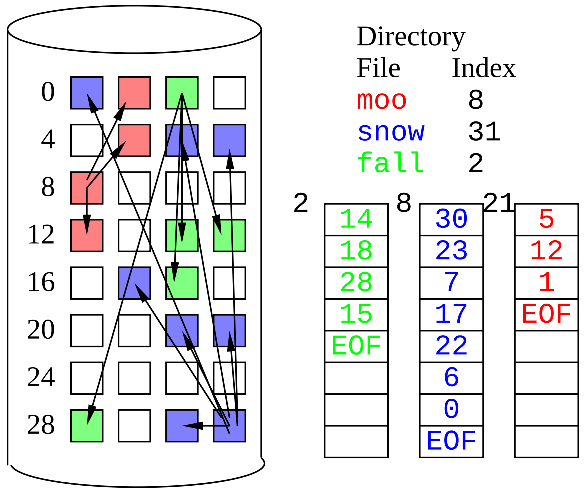
– somewhat better random access – traverse the FAT only – read disk blocks only for the data stored there

– each disk block needs a FAT entry – total number of blocks, in turn total size of a partition – is limited by the size of the FAT

– increased block size means fewer blocks/FAT entries, but more internal fragmentation

- Indexed Allocation

  Use disk blocks as *index blocks* that don't hold file data, but hold pointers to the disk blocks that hold file data.



Directory

File	Index
moo	8
snow	31
fall	2

2	8	21
14	30	5
18	23	12
28	7	1
15	17	EOF
EOF	22	
	6	
	0	
	EOF	

– directory entry now contains a pointer to the index block

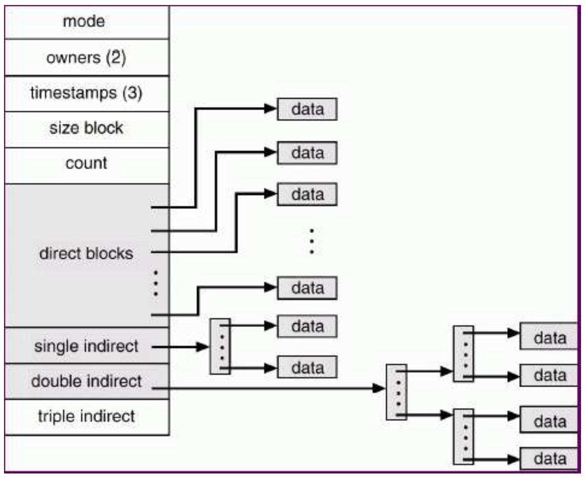
- each file's index block contains pointers to all of its data blocks
- random access is similar to FAT
- a bad data block costs only that block, bad index block could cost the entire file
- size of a file is limited by the number of pointers a data block can hold – if a block holds 512 bytes, and a pointer to a disk block takes 2 bytes, we are limited to 256-block, or 128 KB files
- now even small files require two data blocks – extra disk reads, and potentially wasted space

Can get around the file size limitation in a few ways:

- *linked indexed allocation* – use the last entry in the index block as a pointer to another index block
  - ∗ this removes file size limitations
  - ∗ random access becomes a bit harder
- *two-level index* – the index block points only to other index blocks
  - ∗ file size limitation is not as severe – for example above, disk file now are addressed by a 256-entry index block, each of which points to a 256-entry index block, meaning we can store 65536-block or 32 MB files.
  - ∗ random access is better
  - ∗ but all files take at least 3 blocks of space and access time
- Can add more levels for larger files

- Unix Inodes

  Many Unix filesystems (Berkeley Fast Filesystem, Linux ext2fs, Sun ufs, ...) take an approach that combines some of the ideas above.



  - each file is indexed by an *inode*

- inodes are special disk blocks set aside just for this purpose (see `df -i` to see how many of these exist on your favorite Unix filesystem)
- they are created when the filesystem is created
- the number of inodes limits the total number of files/directories that can be stored in the filesystem
- the inode itself consists of
  - administrative information (permissions, timestamps, etc.)
  - a number of direct blocks (typically 12) that contain pointers to the first 12 blocks of the file
  - a single indirect pointer that points to a disk block which in turn is used as an index block, if the file is too big to be indexed entirely by the direct blocks
  - a double indirect pointer that points to a disk block which is a collection of pointers to disk blocks which are index blocks, used if the file is too big to be indexed by the direct and single indirect blocks
  - a triple indirect pointer that points to an index block of index blocks of index blocks...
- interesting reading on your favorite FreeBSD system: `/sys/ufs/ufs/dinode.h`
- small files need only the direct blocks, so there is little waste in space or extra disk reads in those cases
- medium sized files may use indirect blocks
- only large files make use of (and incur the overhead of) the double or triple indirect blocks, and that is reasonable since those files are large anyway
- since the disk is now broken into two different types of blocks – inodes and data blocks, there must be some way to determine where the inodes are, and to keep track of free inodes and disk blocks. This is done by a *superblock*, located at a fixed position in the filesystem. The superblock is usually replicated on the disk to avoid catastrophic failure in case of corruption of the main superblock

Disk Allocation Considerations:

- limitations on file size, total partition size
- internal, external fragmentation
- overhead to store and access index blocks
- layout of files, inodes, directories, etc, as they affect performance – disk head movement, rotational latency – many unix filesystems keep clusters of inodes at a variety of locations throughout the file system, to allow inodes and the disk blocks they reference to be close together
- may want to reorganize files occasionally to improve layout (disk defragmenting, etc)

Free Space Management

With any of these methods of allocation, we need some way to keep track of free disk blocks.

Two main options:

1. *bit vector* – keep a vector, one bit per disk block
   - 0 means the corresponding block is free, 1 means it is in use
   - search for a free block requires search for the first 0 bit, can be efficient given hardware support
   - vector is too big to keep in main memory, so it must be on disk, which makes traversal slow
   - with block size 2^{12} or 4KB, disk size 2^{33} or 8 GB, we need 2^{21} bits (128 KB) for bit vector (seems reasonable)
   - easy to allocate contiguous space for files
2. *free list* – keep a linked list of free blocks
   - with linked allocation, can just use existing links to form a free list
   - with FAT, use FAT entries for unallocated blocks to store free list
   - no wasted space
   - can be difficult to allocate contiguous blocks
   - allocate from head of list, deallocated blocks added to tail, both $O(1)$ operations
   - Alternative: keep a list of "extents" which is the address of a free block and the number of consecutive free blocks starting there

Performance Optimization

Caching is an important optimization for disk accesses.

A disk cache may be located:

- main memory
- disk controller
- internal to disk drive

For a lecture assignment, you will read about a strategy used by many Unix variants to use main memory as a disk cache: the *buffer cache*.

Safety and Recovery

When a disk cache is used, there could be data in memory that has been "written" by programs, which which has not yet been physically written to the disk. This can cause problems in the event of a system crash or power failure.

If the system detects this situation, typically on bootup after such a failure, a *consistency checker* is run. In Unix, this is usually the `fsck` program, and in Windows, `scandisk` or some variant. This checks for and repairs, if possible, inconsistencies in the filesystem.

Journaling Filesystems

One way to avoid data loss when a filesystem is left in an inconsistent state is to move to a *log-structured* or *journaling* filesystem.

- record updates to the filesystem as *transactions*
- transactions are written immediately to a log and the action is committed by the OS (the application can continue), though the actual filesystem may not yet be updated
- transactions in the log are asynchronously applied to the actual filesystem, at which time the transaction is removed from the log
- if the system crashes, any pending transactions can be applied to the filesystem – main benefits are less chance of significant inconsistencies, and that those inconsistencies can be corrected from the unfinished transactions, avoiding the long consistency check
- Examples:
  - ReiserFS, see `http://www.namesys.com`, a linux journaling filesystem – I recommend reading this page
  - ext3fs, also for linux
  - jfs, see `http://jfs.sourceforge.net/`, IBM journaling filesystem, available for AIX, Linux
  - Related idea in FreeBSD's filesystem: Soft Updates, see `http://www.freebsd.org/doc/en\_US.ISO8859-1/books/handbook/configtuning-disk.html`
  - Journaling extensions to Macintosh HFS disks
  - NTFS does some journaling, but some claim it is not "fully journaled"
- the term "journaling" may also refer to systems that maintain the transaction log for a longer time, giving the ability to "undo" changes and retrieve a previous state of a filesystem

Final Words on File Systems

Things to consider when working with file systems:

- how to partition the disk
- how to organize blocks within a partition to form a file system
- structure of directories
- allocation of space for files
- free space management

RAID

To this point, we have talked about “partitions” as subdivisions of an individual disk. It is also possible to have a logical “partition” span multiple disks.

RAID – *Redundant Array of Independent/Inexpensive Disks*

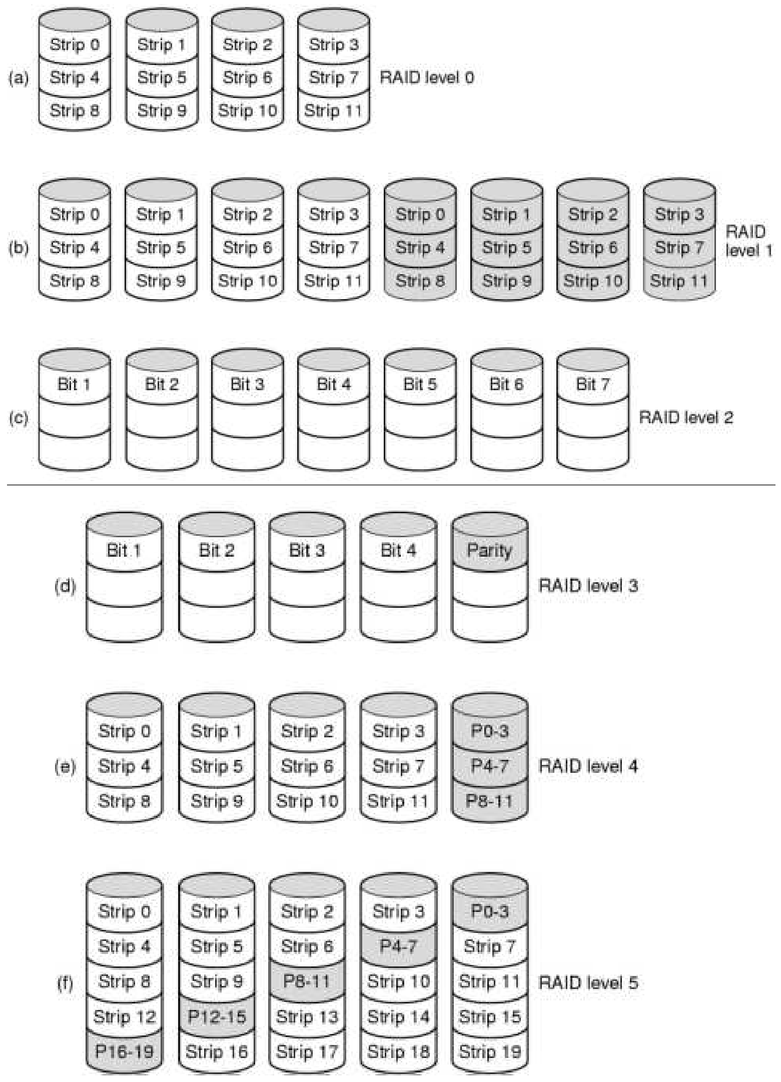
- multiple disks to provide reliability through redundancy
- efficiency – work can be spread across a number of disks or even disk controllers
- convenience of one large partition instead of many small ones

My experience with RAID: the bullpen cluster:

12 disks, 18 GB each. Connected to one Wide-SCSI controller. The system sees it as one big partition:

```
> df -k /export/raid
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c1t5d0s6    191175687 25709437 163554494    14%    /export/raid
```

There are many ways to organize a RAID (Tanenbaum, Figure 5-19):



- RAID Level 0
  - basically just paste together a bunch of disks to see them as one big partition
  - striping
  - not really a RAID, as it is not redundant
  - reliability: one disk failure results in potential loss of entire partition! Actually lower the MTBF (mean time between failures)
  - little or no overhead on writes
  - 100% of disk space is usable for storage
- RAID Level 1
  - mirroring
  - reliability: any failed disk can be reconstructed from its mirror with a simple copy

- all writes must be written to two disks
- reads can come from either of two disks – spread out the load
- 50% of disk space is usable for storage

- RAID Level 2
  - use 7 synchronized disks to store 4 disks worth of data
  - for each 4 bits, compute a 7-bit Hamming-coded (see `http://mathworld.wolfram.com/HammingCode.html` word
  - Hamming codes are self-correcting for one error, can detect two errors
  - 57.1% of disk space is usable for storage
- RAID Level 3
  - like RAID-2, but use a single parity bit
  - still can recover a lost disk using the parity bit
  - two lost disks means entire partition is lost
  - can work with any number of disks
  - can include multiple parity disks
  - space overhead of the parity disk(s)
- RAID Level 4
  - use stripes for parity unit, allowing disks to work independently
  - still can recover a lost disk
  - parity disk can be a bottleneck as all writes require a write to the parity disk
  - space overhead of the parity disk(s)
- RAID Level 5
  - like RAID-4, but parity bit is distributed across all disks
  - This is what we have in bullpen and it has actually worked when disks have failed
- RAID Level 6
  - Like RAID-5 but with error correcting codes
- RAID Level 0+1, 1+0
  - Combine RAID Level 0 (for striping/efficiency) with RAID Level 1 (for redundancy)

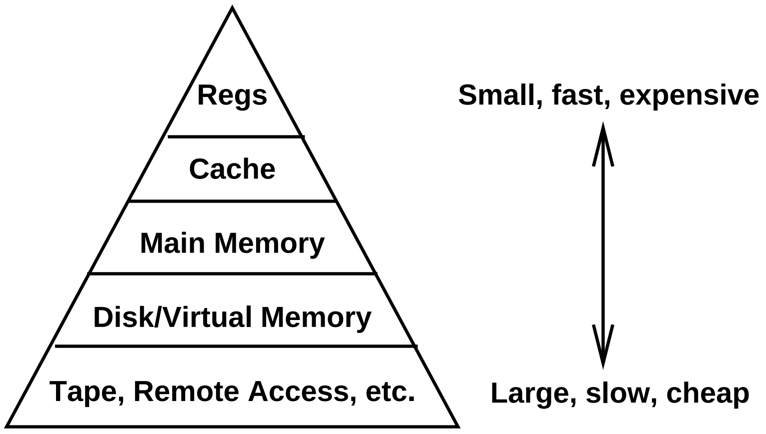
Where does this happen?

- RAID controller – work is done by hardware, OS sees a single drive
- kernel/device driver – work is done by the OS in software – OS uses disks independently, but presents them to "users" as a single unit

In FreeBSD see "vinum" and in Linux see the MD driver and the HOWTO (`http://www.tldp.org/HOWTO/Software-RAID-HOWTO.html`)

Hierarchical Storage

Recall our memory hierarchy:



Just as virtual memory uses disks to simulate a larger main memory, tapes and other removeable media can be used to simulate a larger disk.

- extend the filesystem
- small and frequently-used files remain on disk
- large, old, rarely-used files are archived on tapes
- when one of the old files is requested, the file is brought back onto the disk from the appropriate tape
- usually implemented as a jukebox of tapes or removeable disks
- tape latency is typically 1000 times that of a disk
- add in a tape robot that has to go fetch a tape and it is even worse
- or worse yet, a human who has to be notified, go to the "tape room", find the tape, bring it to the drive, load it

These systems are found at large supercomputing centers.

- HPSS – High Performance Storage System, see `http://www.sdsc.edu/hpss/hpss1.html`
- Unitree, see `http://archive.ncsa.uiuc.edu/SCD/Hardware/UniTree/`

Other issues:

- how to decide when to archive to tape
- retrieval from archive may be fully automated or users may need to explicitly request files from tapes
- duplicate tapes? – tapes can be unreliable
- when is this worthwhile? Can't we just archive information manually?