Topic 13: File Structure

Reading: Chapter 4

- *File*: A named collection of bytes stored on disk.
- From the OS's standpoint, the file consists of a bunch of blocks stored on the device.
- Programmer may actually see a different interface (e.g., byte-stream, records, objects).
  - But this does not matter to the file system (just pack bytes onto disk blocks, unpack them again on reading).

- Disk performance trends:
  - Focus is on capacity, not performance.
  - Smaller — faster RPM, shorter seeks. Net gain is about 7% to 10% per year.
  - Track buffer — store entire track (or more) in RAM.
- Disks are improving at about 10% per year, computers at about 100% every 24 months. What's a file system to do?
  - Use the disk intelligently.
  - Avoid using the disk(!)
- Since it is time-consuming to use the disk, try to go about it intelligently. Use the CPU to figure out how to improve disk access times. Take advantage of locality.
  - Disk scheduling: Sort requests to maximize throughput. Shortest-seek-time, scan, etc. Only effective for long queues of disk requests.
  - File layout: Organize file data on disk so that seek distances are minimized. *Contiguous allocation* is an example of this.
  - File clustering: If files are used together, store them together.

- Common addressing patterns:
  - *Sequential*: Information is processed in order, one piece after the other. This is by far the most common mode. E.g. editor writes out new file, compiler compiles it, etc.
  - *Random Access*: Can address any block in the file directly without passing through its predecessors. E.g., the data set for demand paging (virtual memory), databases.
  - *Keyed*: Search for blocks with particular values. E.g., hash table, associative database, dictionary. Usually not provided by a general purpose operating system.
- Modern file systems must address four general problems:
  - *Disk Management*: Efficient use of disk space, fast access to files, sharing of space between several users.
  - *Naming*: How do users select files?
  - *Protection*: All users are not equal; all users do not have access to all files.
  - *Reliability*: Information must last safely for long periods of time.
- Each file is represented by a *file record* stored on the disk. This structure contains information about the locations of the file's blocks, and *attributes* for the file. Attributes include: file size, owner/group, protection information, create/modify/access times, reference count (how many names the file has), etc. Note that the file name is not an attribute if a file can have more than one name.
  - The information stored about the files (e.g., block locations, attributes, names) is referred to as *metadata*.

File Operations:

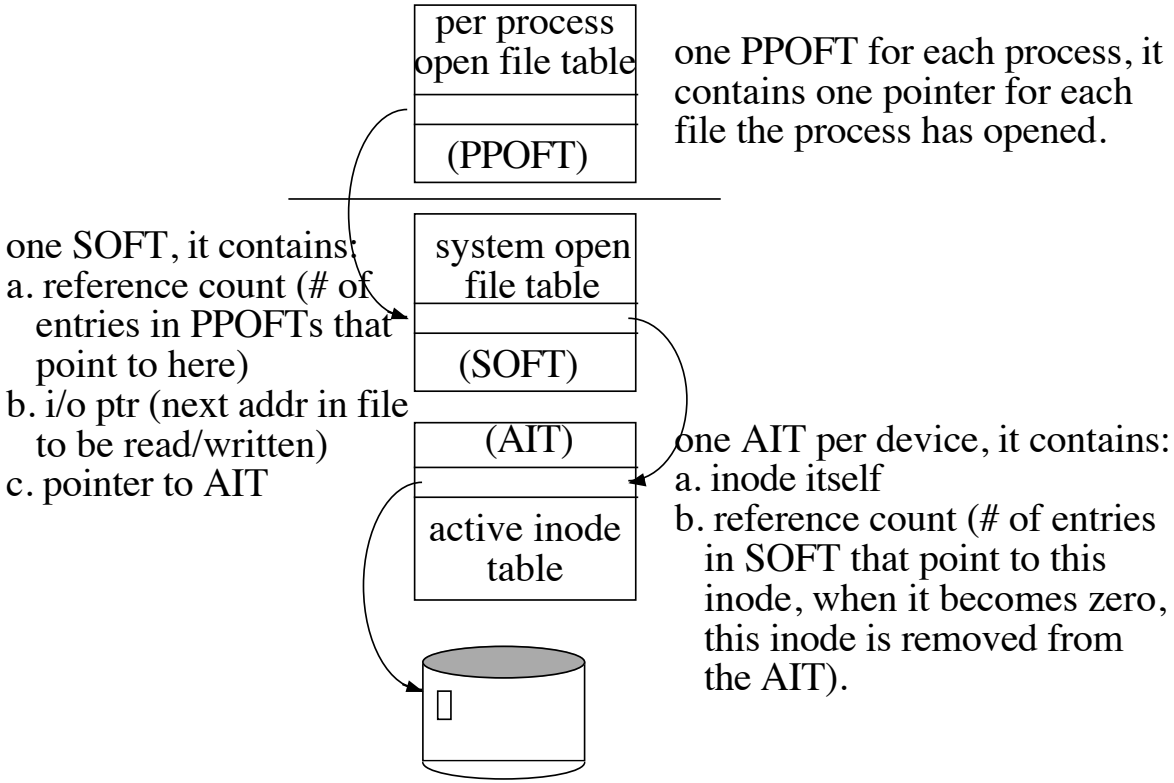
- Create: Allocate space, create name.
- Delete: Remove name, reclaim space *if* reference count is zero.
- Open: Use name to find file (more on this later) and return *file handle* or *file descriptor*. In UNIX, a file descriptor is simply an integer that is used to index into a per-process *file descriptor table* in the kernel that contains information about the open file.
- Read/Write: Read from or write to an open file specified by a file descriptor. Each open file has an associated *offset*; read and write implicitly use and update the offset.
- Seek: Explicitly changes the offset (location of next read or write) for an open file.
- Close: Removes the entry in the file descriptor table.

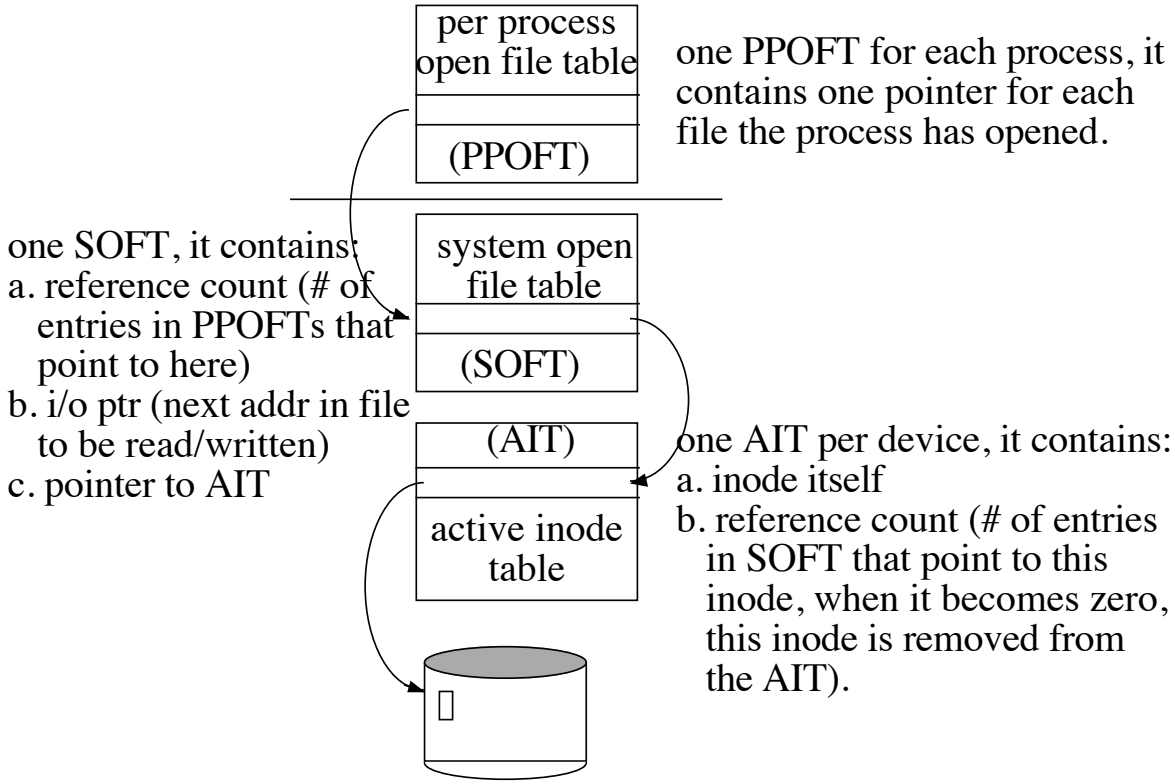
UNIX difficulties:

- File descriptors may be duplicated.
  - This is useful for redirection: e.g.,

    ```
    wc < foo.c
    ```

- After a fork, both processes share the same offset for each open file.
  - Offsets cannot be stored in the file descriptor table.
  - UNIX also has an open file table which stores the offsets and solves the problem.
- UNIX provides a uniform set of operations on a variety of objects, including files, pipes, and devices. This means that there may be different data structures “below” the open file table.



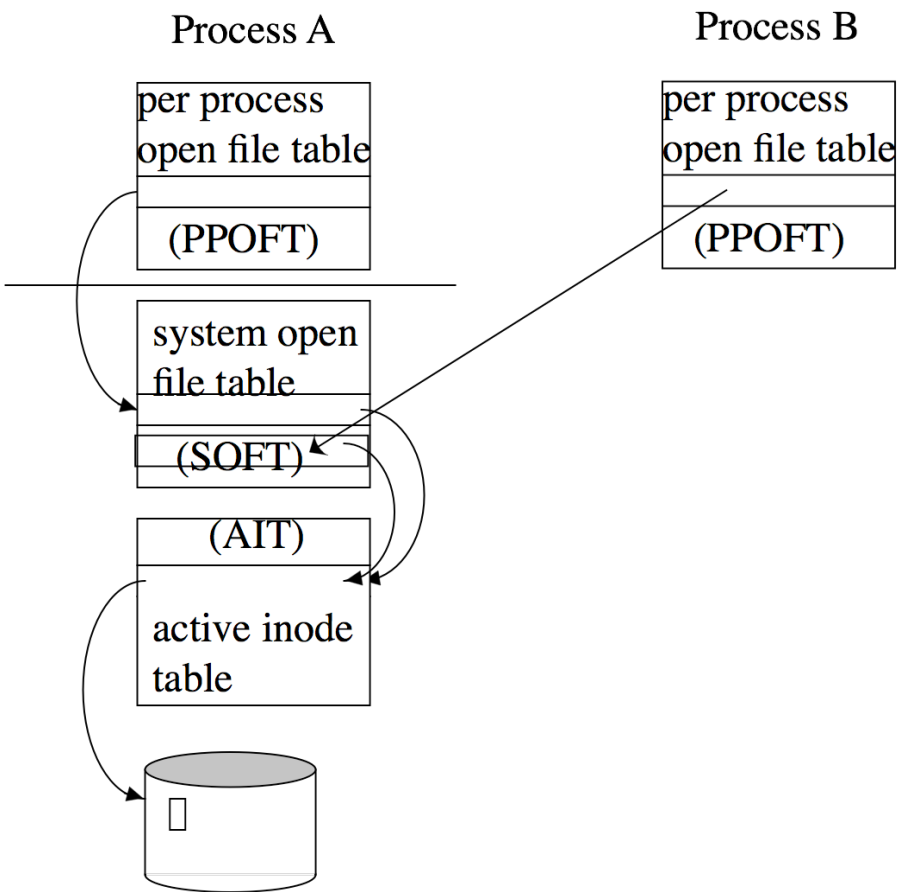

per process
open file table
(PPOFT)
one PPOFT for each process, it
contains one pointer for each
file the process has opened.
one SOFT, it contains:
a. reference count (# of
entries in PPOFTs that
point to here)
b. i/o ptr (next addr in file
to be read/written)
c. pointer to AIT
system open
file table
(SOFT)
(AIT)
active inode
table
one AIT per device, it contains:
a. inode itself
b. reference count (# of entries
in SOFT that point to this
inode, when it becomes zero,
this inode is removed from
the AIT).

Two processes independent of each other (not in the other process's tree).

Each need a separate count of where they are in the file.

There are 2 entries in SOFT referring to the same entry in AIT. Need a reference count in AIT.

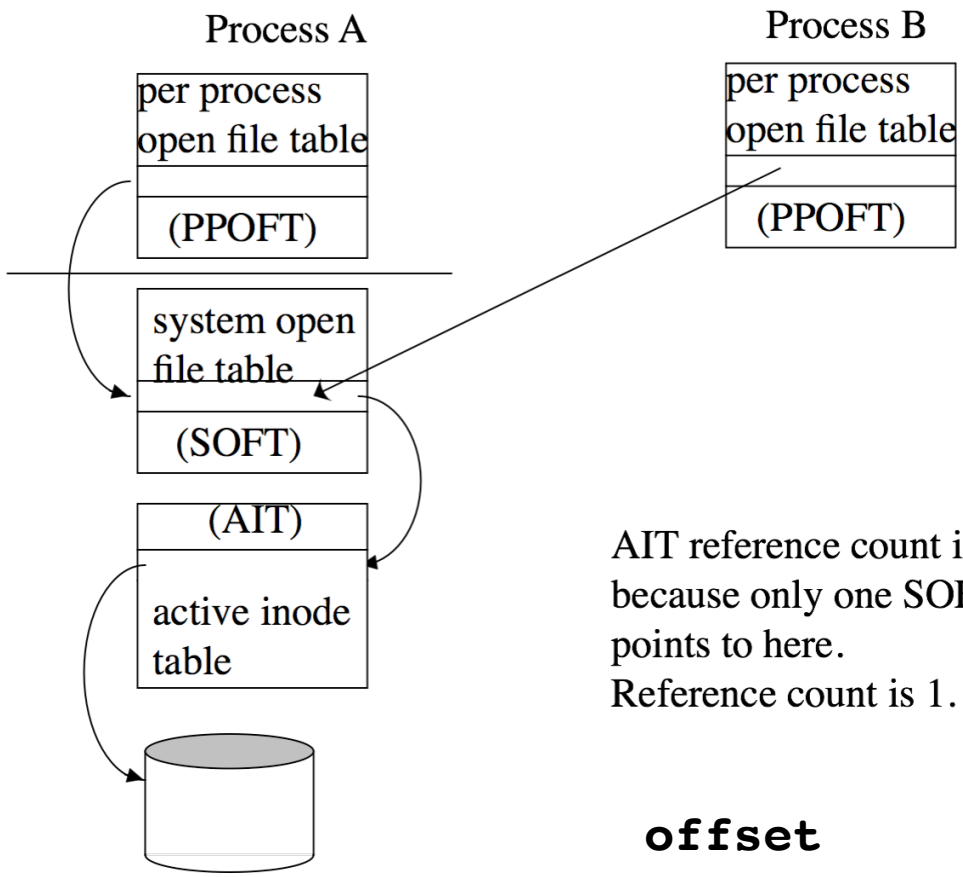
AIT reference count = number of SOFT entries that point to here.



offset

cnt=1

Process A opens a file, reads 5000 bytes from it.
Process A then does a fork to create Process B



SOFT keeps a reference count of the number of PPOFT's that refer to this SOFT entry.

Count is now 2.

AIT reference count is 1 (not 2) because only one SOFT entry points to here.
Reference count is 1.

```
kidpid = fork();
if (kidpid == 0)
    execl(...);
else
    zap(...);
```

```
 offset
A does lseek(10000);
        cnt=1
```

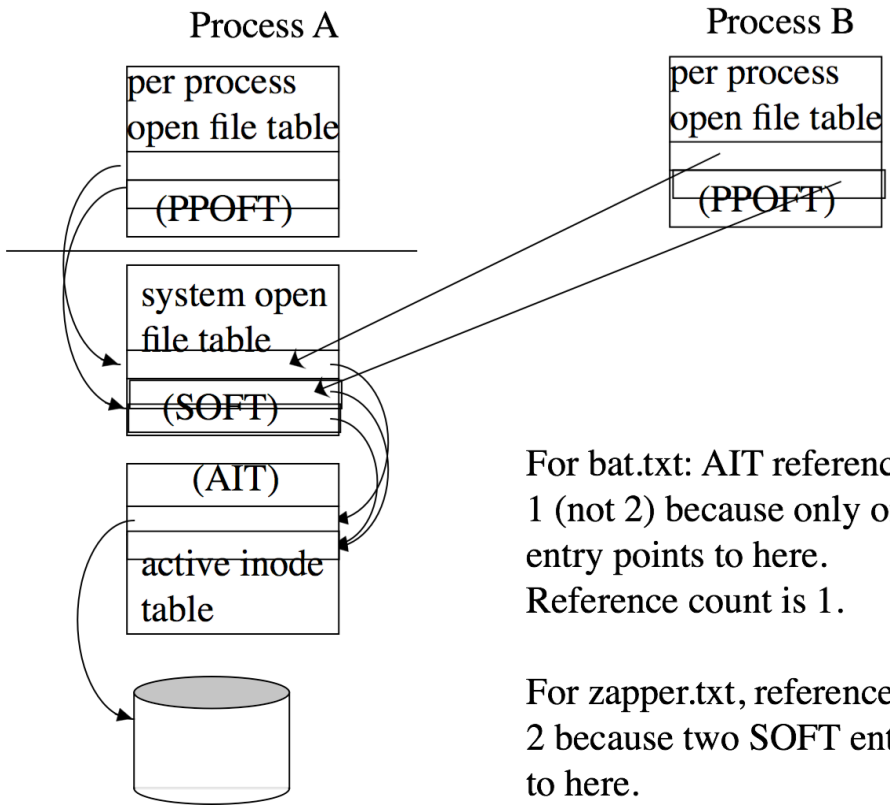
Process A opens a file, bat.txt, then does a fork to create Process B.

Process B then opens a file named zapper.txt.
Process A then opens zapper.txt

SOFT keeps a reference count of the number of PPOFT's that refer to this SOFT entry.

Count is now 2 for bat.txt.

Count for zapper.txt is 1 for the entry for B, and is 1 for the entry forA.



For bat.txt: AIT reference count is 1 (not 2) because only one SOFT entry points to here.
Reference count is 1.

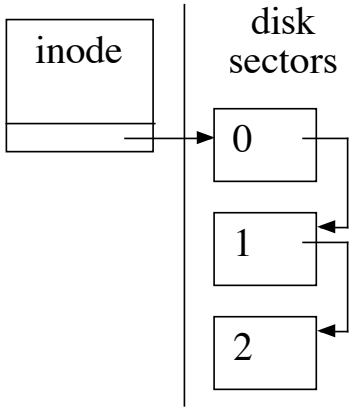
For zapper.txt, reference count is 2 because two SOFT entries point to here.

- Disk Management: how should the disk sectors be used to store the blocks of a file? Each file record describes the file's sectors. File records are stored on the disk along with the files they describe; more on this later.

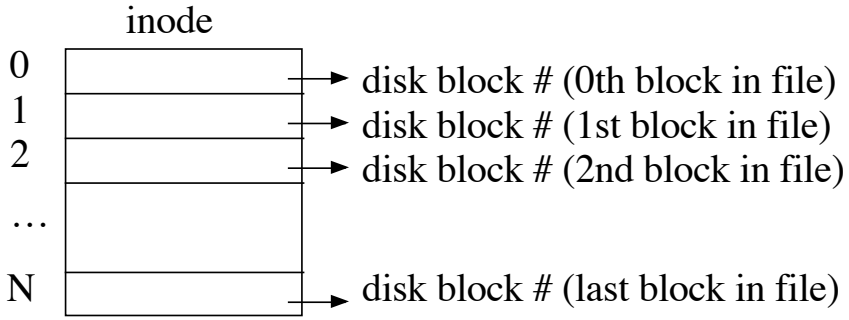
- Things to think about when deciding how to lay files out on the disk:
  - Most files are small — often smaller than one disk sector.
  - Much of the disk is allocated to large files.
  - Many of the I/O operations are made to large files.
  - Thus, per-file cost must be low, but large files must have good performance.
  - Most files are accessed sequentially; most files are accessed in their entirety through a sequence of sequential operations.

- *Contiguous allocation*: Allocate files like segmented memory (give each disk sector a number from 0 up). Keep a free list of unused areas of the disk. When creating a file, make the user specify its length, allocate all the space at once. Descriptor contains location and size.
  - \+ Advantages: Easy access, both sequential and random. Simple. Minimizes seeks.
  - \- Drawbacks: Horrible fragmentation will make large files impossible, unless you compact. Hard to predict space needs at file creation time.
  - Examples: IBM OS/360, Bullet.

- *Linked List Allocation*: Keep a linked list of all free blocks. In the file descriptor, just keep pointer to first block. In each block of the file, keep a pointer to the next block of the file.
  - \+ Advantages?
  - \- Drawbacks?



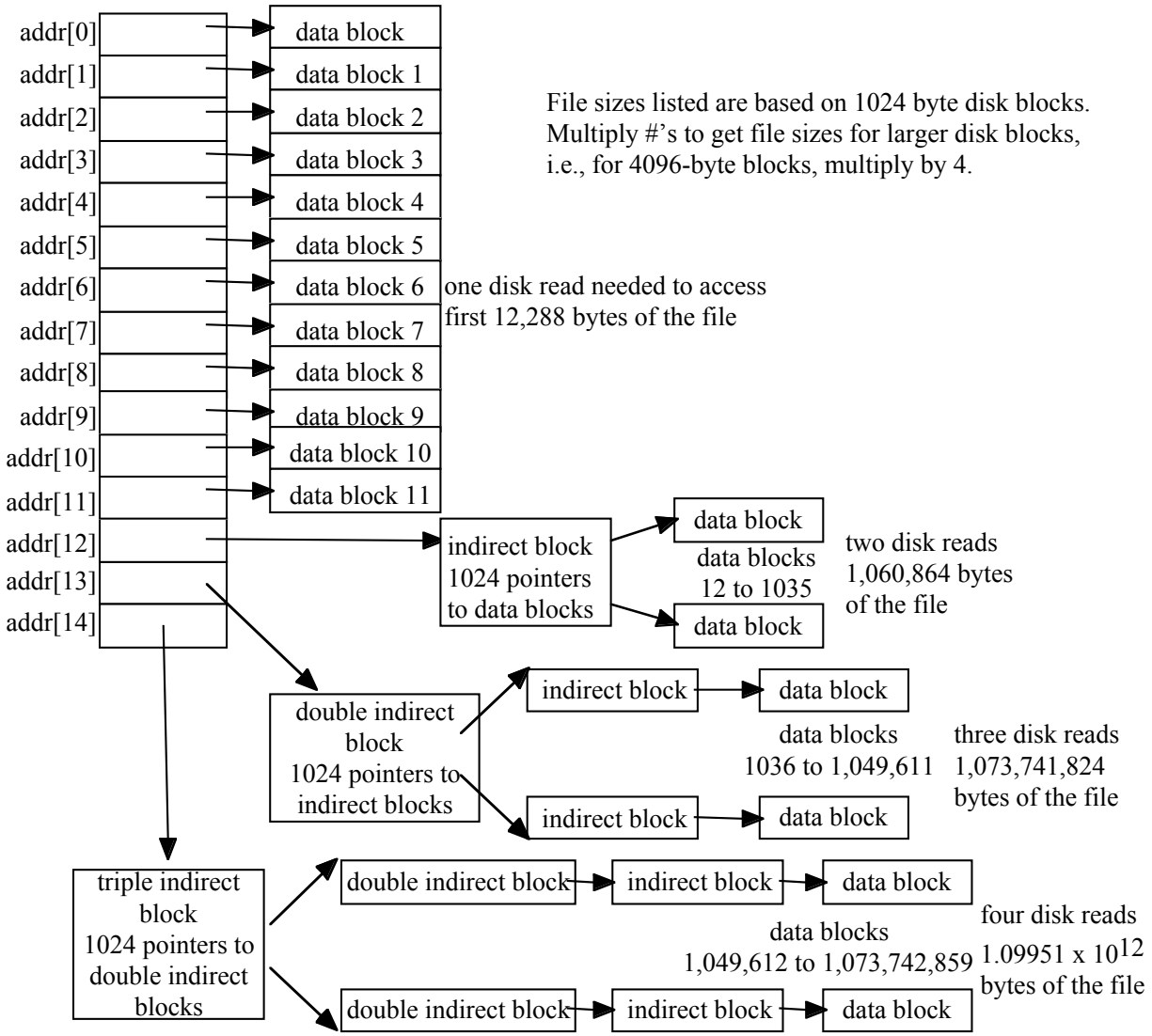
- *Indexed files*: Keep an array of block pointers for each file. File maximum length must be declared when it is created. Allocate an array to hold pointers to all the blocks, but do not allocate the blocks. Then, fill in the pointers dynamically using a list of free blocks.
  - \+ Advantages? Not as much space wasted by over-predicting, both sequential and random access are easy.
  - \- Disadvantages? Still have to set maximum file size, and there will be lots of seeks.



- *Multi-level indexed files*: 4.3 BSD solution:
  - File record is called an *i-node* (index node) and contains 15 block pointers. The first 12 point to data blocks, the next one to an indirect block (contains 1024 more block pointers), then a double-indirect block, and finally a triple-indirect block. Maximum file length is fixed, but is large. Descriptor space is not allocated until needed.
  - Examples: block 23, block 5, block 1040.
  - \+ Advantages: simple, easy to implement, incremental expansion, easy and fast access to small files.
  - \- Potential drawbacks:
    - Indirect mechanism does not provide very efficient access to large files: up to 3 descriptor accesses for each real read/write operation.
    - Block-by-block organization of free list may mean that file data gets spread around the disk.
- Keeping track of free blocks in Unix: use a *bit map*.
  - Just an array of bits, one bit per disk block. 1 means "block free", 0 means "block allocated". For a 120-GB disk with 4 KB sector size:

$$\frac{120 \times 2^{30}}{4 \times 2^{10}} = 30 \times 2^{20} \text{ disk sectors}$$

$$\frac{30 \times 2^{20} bits}{2^3 bits/byte} = 30 \times 2^{17} \approx 2^5 \times 2^{17} = 2^{22} = 4 \text{ MB}$$


addr[0]
addr[1]
addr[2]
addr[3]
addr[4]
addr[5]
addr[6]
addr[7]
addr[8]
addr[9]
addr[10]
addr[11]
addr[12]
addr[13]
addr[14]
data block
data block 1
data block 2
data block 3
data block 4
data block 5
data block 6
data block 7
data block 8
data block 9
data block 10
data block 11
File sizes listed are based on 1024 byte disk blocks.
Multiply #'s to get file sizes for larger disk blocks,
i.e., for 4096-byte blocks, multiply by 4.
one disk read needed to access
first 12,288 bytes of the file
indirect block
1024 pointers
to data blocks
data block
data blocks
12 to 1035
data block
two disk reads
1,060,864 bytes
of the file
double indirect
block
1024 pointers to
indirect blocks
indirect block
data block
data blocks
1036 to 1,049,611
three disk reads
1,073,741,824
bytes of the file
indirect block
data block
triple indirect
block
1024 pointers to
double indirect
blocks
double indirect block
indirect block
data block
data blocks
1,049,612 to 1,073,742,859
four disk reads
1.09951 x 10^12
bytes of the file
double indirect block
indirect block
data block

- Improving performance:
  - Disk scheduling.
  - Caching.
  - Optimized block allocation:
    - During allocation, try to allocate the next block of the file "close" to the previous block of the file. If the disk is not full, this will usually work well.
    - Unix FFS (Fast File System): blocks of the same file are allocated from the same *cylinder group*.
    - If the disk becomes full, this becomes VERY expensive without getting much in the way of adjacency. Solution: keep a reserve (e.g., 10% of disk), and do not even tell users about the reserve. Never let the disk get more than 90% full.
    - With multiple surfaces on the disk, there are multiple optimal next blocks; with 10% of the disk free, can almost always use one of them.
- Unix FFS disk layout:
  - *Superblock* — contains file block size, number of blocks, cylinder group size
  - Each cylinder has a copy of the superblock, stored in different locations within the cylinder.
    - Each cylinder also has a *cylinder block*:
      - Number of i-nodes.
      - Bit maps for blocks.

- Better yet, do not use the disk. Instead, *cache* file data in main memory. This is called a *block cache*.
  - Store popular blocks in memory, eliminating disk accesses entirely.
    - Short-lived file data can be filtered out completely.
  - Hide disk latency by overlapping computation with I/O.
    - *Read-ahead* blocks that will be read.
    - *Write-behind* blocks that were written.
    - Read-ahead and write-behind increase queue length, which in turn makes disk scheduling effective.
- Disadvantages of a file cache:
  - Requires RAM.
  - Contents are lost in a crash. This is a problem for blocks that are written to the cache but not to the disk. There are several solutions:
    - *Write-through*: write blocks to disk and cache at the same time.
    - *Write-through-on-close*: improves performance of applications that write the same blocks many times.
    - *Bounded write-back*: limit the length of time new data can live in cache before being written to disk. 30 seconds is the Unix standard.
  - *Cache consistency*: an inconsistency between the cache contents and the disk can lead to stale data errors.