Introduction to File Systems

CSE 120
Spring 2002

Files

Files are an abstraction of memory that are *stable* and *sharable*.

Typically implemented in three different layers of abstraction

3 *I/O system*: interrupt handling, scheduling.

2 *Physical file system*: containers $\rightarrow$ physical blocks .

1 *Logical file system*: paths $\rightarrow$ containers.

Stream-based files

```
int open(char *path, int oflag)
int close(int fildes)
int read(int filedes, void *buf,
  int nbyte)
```
returns number actually read

```
int write(int filedes, void *buf,
  int nbyte)
```
returns number actually written

```
long lseek(int fildes, long
  offset, int whence)
```

The need for caching/read-ahead

Given a disk with 4KB blocks:

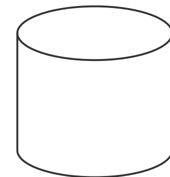
- read metadata 15×10^{-3} sec seek
  $4 \times 10^{3}/10^{7}$ sec xfr
- read data + 15 msec
- total xfr time 3×10^{-2} sec
- transfer rate 4×10^{3} B/3×10^{-2} sec
  = 133 KB/sec
  (vs. 10 MB/sec)

File Systems vs. VM Systems

Logical File System

Physical File System

Driver

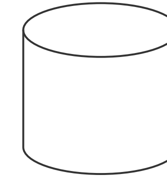


Segment Naming

Page Aging

Swapping

Driver



```
d1 = open(file);
read(d);
compute a bit.
write(d);
close(d);
```

Mapped files

```
void *mmap(void *addr, long len,
  int prot, int flags, int fildes,
  long off)
int munmap(void *addr, long len)
```

Memory map example

Write the first 1,024 characters of file `arg[1]` the character in `arg[2]`.

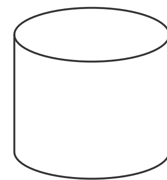
```
fd = open(argv[1], O_RDWR|O_CREAT, 0666);
data = (char *)
  mmap(0, 1024, PROT_READ|PROT_WRITE,
  MAP_SHARED, fd, 0));
for (i = 0; i < 1024; i++)
  data[i] = *argv[2];
munmap(data, 1024);
```

The (original) Unix file system

Logical File System:
open, close, read, write

Physical File System:
caching of data and metadata

Driver



Two caches:

- disk block cache
- metadata (*inode*) cache

... implemented as a hash table with LRU replacement of unlocked values.

Locking values into the cache

When should a value be locked into the cache?

- disk cache: for the duration of the systems call.
- metadata cache: for the time that the file is open.

inode cache issues

- A Unix file is deleted when there are *no links* to the file.

  Ex: `echo 123 > f; ln f b; rm f; rm b`

  Consider

  ```
  f = creat("foo");
  unlink("foo");
  write(f, ...);
  ```

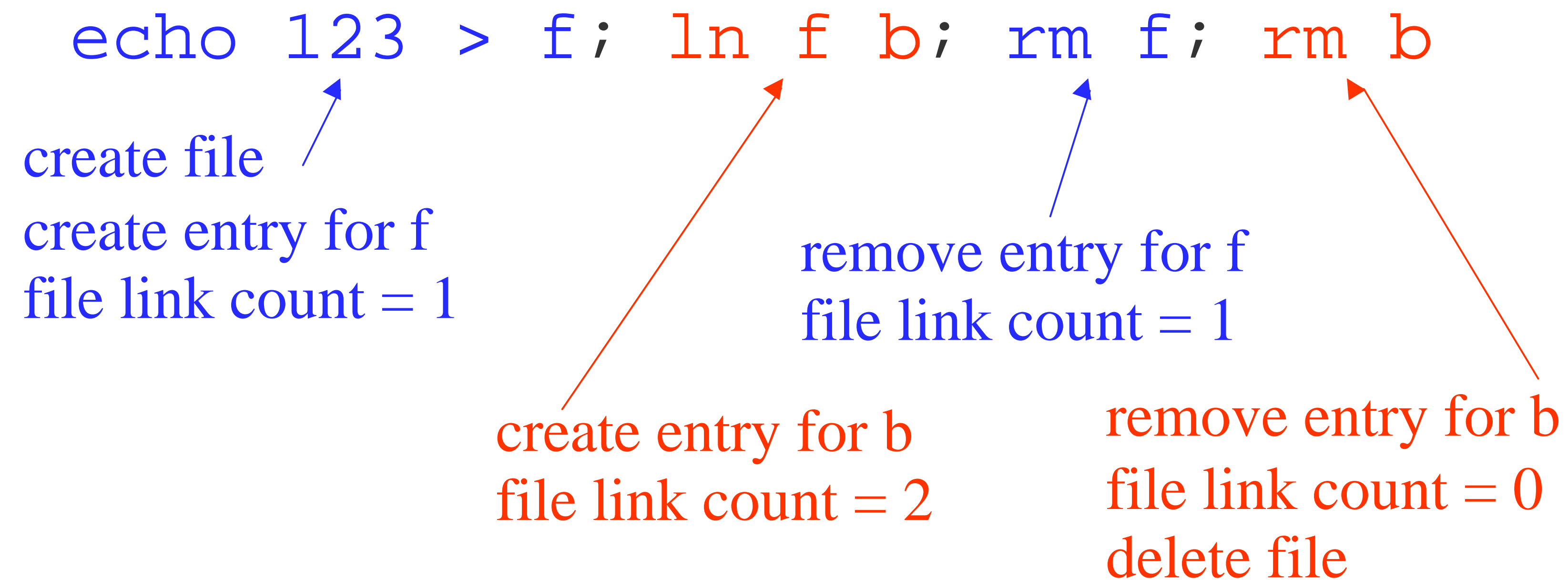
  ... what now?

- Can locking in cache lead to deadlock?

inode cache issues II

Caching metadata can cause problems with respect to crashes.

example: link count



Reliability-induced synchronous writes

Safety: for all files *f*: always

(number links to *f*) =

(link count in *f*'s metadata)

How can this be implemented in the face of processor crashes?

R-i synchronous writes II

(number links to f) $\leq$
(link count in f's metadata)

(number links to f) $\geq$
(link count in f's metadata)

Summary so far

- Caching and read-ahead are vital to file system performance.
- Both techniques are important for physical data and for metadata.
- Metadata consistency is an issue both for correctness and for performance.

Containers

The physical file system layer translates *logical block addresses* of a file into *physical block addresses* of the device.

The design of this mapping mechanism becomes more critical as the size of the physical device increases.

- container name $\rightarrow$ list of addresses
- representation of free blocks

Threaded file system

- Use a bitmap to denote a disk block being allocated or free.
- Container name is address of first block of file.
- Each block contains pointer to next block in file (or a special value indicating the last block in file).

Ex: 8G disk (2^{33} B), 1K block, 2^{23} blocks/disk

bitmap: 2^{10} blocks (1K blocks)

pointer: 3 or 4 bytes/block.

File allocation table

Improve random access by localizing pointers: File Allocation Table.

Container name is first block address.

Free list is also stored in FAT.

3	0	1	6
7	4	8	0
...			

free list: 3, 6, 8, ...
file 2: 2, 1
file 5: 5, 4, 7

FAT space requirements

$|\text{FAT}| = 2^f$ bytes

$|\text{disk}| = 2^d$ bytes or 2^{d-b} disk blocks

$|\text{block}| = 2^b$ bytes addressed with $d-b$ bits

---

so $2^{f+3} \geq (d-b)2^{d-b}$

$$f + 3 \geq \log(d-b) + d - b$$

$$b \geq \log(d-b) + d - f - 3$$

---

example 32G disk (d=35), 1M FAT (f=20): smallest b is 17 or 128KB disk block

example 32G disk (d=35), 4K block (b=12): f is 25 or 32M FAT

Unix File System

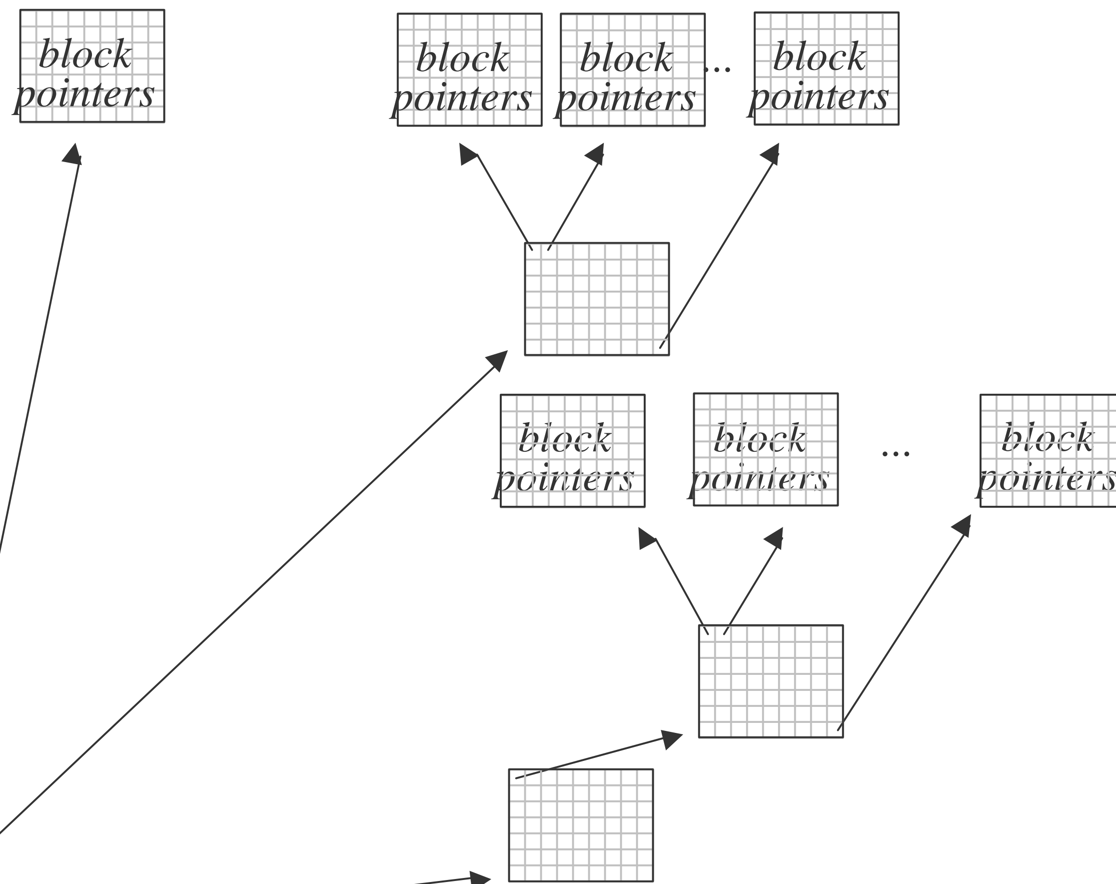
Need to have a large FAT that exhibits better locality. Can be done by having a FAT for each file.

- file system descriptor
- free inode list
- free disk block list

container is index into inode array (*inumber*)

inode

owner id
group id
type (0=free)
permissions
access times
number of links
file length (bytes)
direct pointer 1
direct pointer 2
...
direct pointer 10
indirect pointer 1
indirect pointer 2
indirect pointer 3



Free list management

- Finding a free inode is (relatively) simple: can search list for a inode of type 0.
- Finding a free disk block is (relatively) hard:it is free only if it is not linked in a file.

... first problem allows more leeway for trading off accuracy for performance.

Free disk blocks

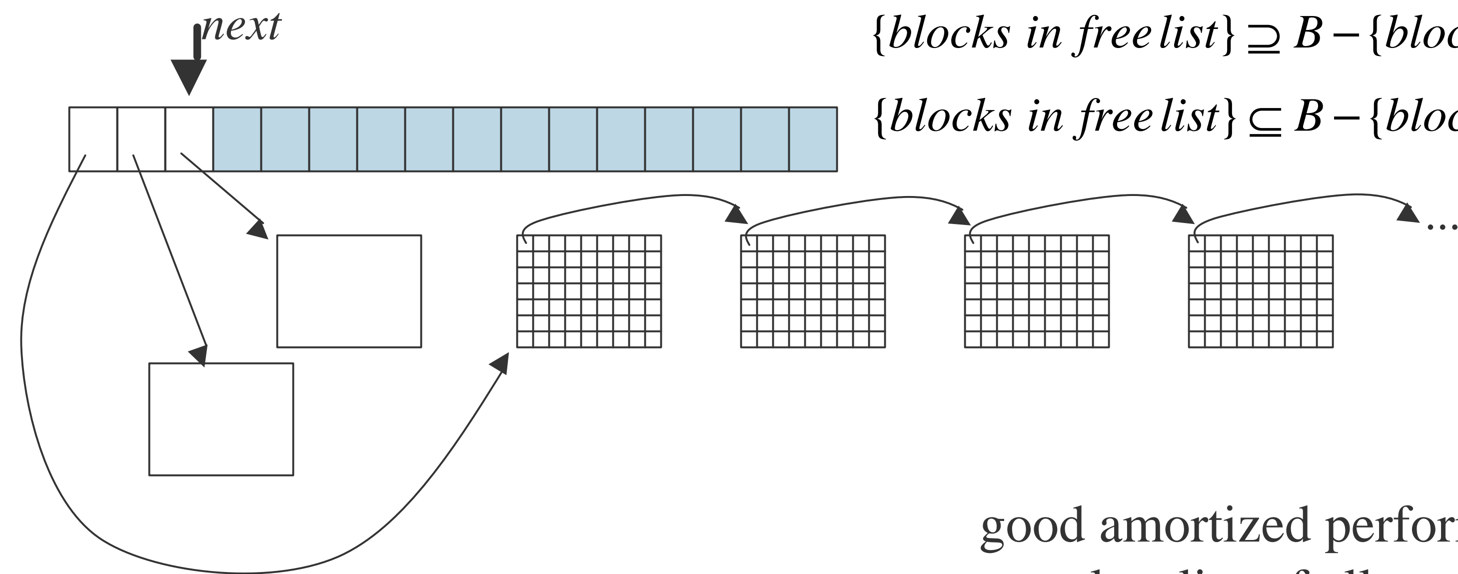
Use free blocks to store the list of free blocks.

Superblock contains block's worth of free disk block pointers.

$$\{blocks\ in\ free\ list\} = B - \{blocks\ in\ files\}$$

$$\{blocks\ in\ free\ list\} \supseteq B - \{blocks\ in\ files\}$$

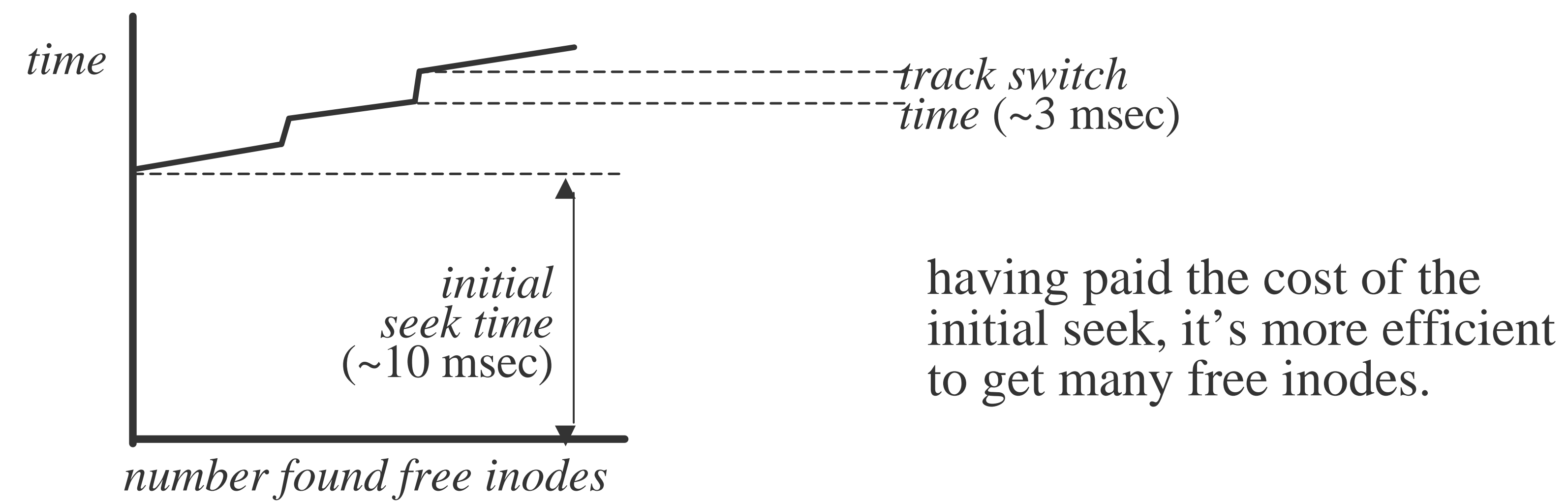
$$\{blocks\ in\ free\ list\} \subseteq B - \{blocks\ in\ files\}$$



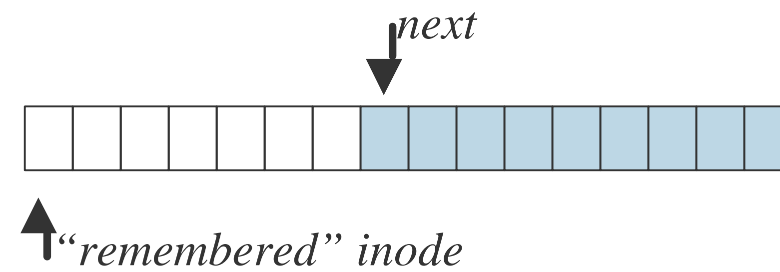
good amortized performance

poor locality of allocated blocks

Free inodes



Free inodes, II




```
void ifree (inumber i) {
   increment free inode count;
   if (superblock locked) return;
   if (inode list full)
      remembered = min(remembered, i);
   else store i in free inode list;
   }
```

Other Unix file systems syscalls

```
int chdir(char *path);
int chroot(char *path);
int link(char *source, char *target);
int unlink (char *path);

int mkfifo(char *path, int mode);
```

- open blocks for read/write rendesvous
- read on empty pipe blocks until no writers
- write on pipe with no readers raises SIGPIPE