

Homework 6

Solution Key

Problem 6.1

Some operating systems provide a system call `rename` to give a file a new name. Describe some differences between using this call to rename a file and just copying the file contents into a new file with the desired name, followed by deleting the old file. (There are at least two major ones.) Are there tradeoffs, or is one technique always better than the other? Does the answer change if the underlying filesystem is contiguous, linked list, FAT, or inode-based? Why or why not?

The two biggest issues are that the copy/delete version will take longer (because it needs to actually read and write the entire file) and break hard links. Another difference is that the copy/delete version will always work, even if the move crosses filesystems—to move a file by renaming, it has to remain on its original system.

While the renaming version is usually going to be preferred when it's possible, there might be times when a user *wants* to break a hard link: if changes are to be made in this one copy of the file that shouldn't be reflected in other copies of the file, it's important to ensure that the files actually be separate and not just links of each other.

Incidentally, one thing that is not different about the techniques is what they do to symbolic links: *both* versions break symlinks, because those are based only on the name of the file.

As for how the question applies to different kinds of filesystems, the “it takes longer” problem certainly applies to all of them. Hard links tend not to be supported in the first place on non-inode systems, but if they somehow did, copy/delete would break them anywhere, since crucially, copy/delete creates a wholly separate file and has no way of communicating the change to other dir entries that point to a file.

Problem 6.2

A system with 512-byte blocks is implemented with a free space bitmap. The beginning of a free space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest-numbered block, so after writing file A, which is 2,871 bytes, the bitmap looks like this: 1111 1110 0000 0000 0000. Show the bitmap after each of the following additional actions. . . Also indicate which blocks are allocated to each file at each point. Document any assumptions you had to make about the system in solving this problem.

I'm assuming that we're using a system that permits noncontiguous file allocation, but other than that it should apply to any system.

Initial values:

```
1111  1110  0000  0000  0000
*AAA  AAA-  ----  ----  ----
```

- a. *File B is created and written, with 2,217 bytes.*

That's five blocks:

```
1111  1111  1111  0000  0000
*AAA  AAAB  BBBB  ----  ----
```

- b. *File A is extended to 3,123 bytes.*

That's just a smidge over $6 * 512$, so we need a seventh block for A:

```
1111  1111  1111  1000  0000
*AAA  AAAB  BBBB  A---  ----
```

- c. *File B is extended to 2,489 bytes.*

That's still a bit under $5 * 512$, so we don't need to allocate any more blocks, and the table is unchanged:

```
1111  1111  1111  1000  0000
*AAA  AAAB  BBBB  A---  ----
```

- d. *File A is deleted.*

Frees up all the A blocks without affecting B:

```
1000  0001  1111  0000  0000
*---  ---B  BBBB  ----  ----
```

- e. *File C is created and written, with 4,003 bytes.*

Just shy of $8 * 512$, so eight blocks will do it:

```
1111 1111 1111 1100 0000
*CCC CCCB BBBB CC-- ----
```

- f. *File B is deleted.*

Leaves a hole in the middle of C:

```
1111 1110 0000 1100 0000
*CCC CCC- ---- CC-- ----
```