Autumn 2020 CS110 Self-Assessment: Filesystems

Provide written answers to each of the five self-assessment questions. You’re free to use any static resources (websites, textbooks, my lecture slides, your own notes), and you’re free to write and compile any test code you’d like if it helps you better respond to the questions. You may not, however, converse with anyone else about these problems.

Each problem will be awarded 0, 1, 2, 3, or 4 points. Any uncontroversially serious response will get at least 1 point, and we expect the vast majority of problem scores to be 3’s and 4’s. The grading criteria will be generous so we can emphasize learning without fussing over the minutiae of scores.

Your final submission is expected to be uploaded to Gradescope, and we ask that your submission be five pages, one question/answer per page. You can edit this PDF, or you can type your answers into a simple text editor and generate a PDF from that. Your submission must be uploaded to Gradescope, without exception, by 8:30am on Tuesday, October 6, Stanford time. If you visit Gradescope and CS110 isn’t your list of courses, you can enroll using access code RWXKZ7.

Problem 1 [4 points]
Lab Handout 01, Problem 3.2 discusses the \texttt{link} system call and its ability to create additional pathnames for existing files, and the Lab Solution 01 handout outlines how \texttt{link} might be—and actually is—implemented for most Unix file systems.

Here’s another idea: Instead of the design outlined in the provided solution, which maps the new pathname to the exact same inode \texttt{number} of the old pathname, we could associate the new pathname with a \texttt{different} inode that’s a shallow snapshot—that is, a simple \texttt{memcpy}—of the one associated with the old pathname. The snapshot is shallow in the sense that the contents of the new inode are an exact replica of the old one, but the payload blocks themselves aren’t replicated. Any block numbers appearing in the first inode are replicated in the second, verbatim. Explain why small changes to the first file—e.g. appending a few additional bytes—might be reflected in the second, but more dramatic changes to the first might \textbf{not} be fully reflected in the second.
Problem 2 [4 points]
The `read` system call will only return 0 when the current supply of bytes has been exhausted and it's clear to the operating system that no more bytes will ever become available.

- When a file descriptor is generated by opening a valid file, as with `open("input.txt", O_RDONLY)`, what information in the relevant open file table entry and relevant vnode table entry would need to be examined in order to determine whether 0 should be returned or not?

- You can coerce a single call to `read(STDIN_FILENO, buffer, sizeof(buffer))` to return 0 by typing ctrl-D. However, typing ctrl-D coerces the current `read` call to return without coercing subsequent calls to `read` to automatically return. That’s means you need to type ctrl-D **twice** to get the short program you see below to run to completion. Restated, ctrl-D is just a way to get you out of the current `read` call, and you don’t truly shut down standard input.

Explain how typing ctrl-D might interact with the relevant open file and relevant vnode table entries so that `read` knows that returning 0 is the correct thing to do? (Understand that there are many reasonable approaches here.)

```c
int main(int argc, char *argv[]) {
    char ch;
    assert(read(STDIN_FILENO, &ch, 1) == 0);
    assert(read(STDIN_FILENO, &ch, 1) == 0);
    return 0;
}
```
Problem 3 [4 points]
Lab Handout 01, Problem 3.4 introduces the idea of a symbolic link (also known as a soft link) and how the pathname-to-inumber resolution algorithm would need to change to allow symbolic links. What Problem 3.4 doesn’t discuss is how the Assignment 2 filesystem itself could be updated to support those symbolic links. Assuming that symbolic links are named directory entries, just like regular files and directories are, and symbolic link names are limited to 14 characters, but that they can be associated with a string of any length. Describe the changes you’d need to make to the inode and file layers of your Assignment 2 codebase to support symbolic links.
Problem 4 [4 points]
A fully operational Assignment 2 filesystem only needs one payload block be allocated for 512-byte files but requires two payload blocks be allocated for a 513-byte file (thereby wasting 511 bytes of disk space). Describe how you might optimize the filesystem implementation to avoid the allocation of an additional payload block when the file size is just a few bytes larger than some multiple of 512.
Problem 5 [4 points]
Your Assignment 2 filesystem could be optimized to support directories with a very large number—large meaning thousands or more—of entries by sorting a directory’s `struct dirent6` in such a way that some binary search for a named entry is easily supported to run in logarithmic time. One idea is to store all `struct dirent6`’s as you already do, except mandate that they be stored in sorted order. Another approach would be to organize the `struct dirent6`’s using a B-tree, where each 512-byte block of payload stores up to 28 `struct dirent6`’s and up to 29 child `uint16_t` block numbers. Describe which of the four layers—pathname, directory, file, and/or inode—would need to change to support either of these, and compare and contrast the two ideas, identifying one relative strength and one relative weakness for each. You’re of course welcome to Google what a B-tree is, but correct answers to this question can simply rely on the fact that a B-tree is an elaborate variation on a balanced binary search tree.