Operating Systems – spring 2022
Tutorial-Assignment 9

Instructor: Hans P. Reiser

Question 9.1: File System Implementation

a. What are hard links?

Solution:
Each directory entry is a hard link; hard links simply map a name to an inode. Having created a file f and a hard link l to it, f and l are indistinguishable. It is useless to try to argument that l is a hard link to f; rather both names f and l are hard links to the file represented by the inode both of them map to.

Note: Hard links are thus implemented through the actual file system, not on the higer virtual file system (VFS) layer. A hard link is possible only within a single actual file system, not across moint points of different file systems.

inodes (index nodes) are blocks managed by a file system implementation that contain all metadata about a file (such as ownership, permissions, timestamps, and location of the file data on disk (details depend on the implementation).

b. What are symbolic links?

Solution:
Symbolic links are files of type "symbolic link". Their content is interpreted by the VFS layer and consists only of the relative or absolute path of the "pointed to" file. For symbolic links, stating that l is a (symbolic) link to f is fine: l is the link, its contents is "f". Differing from hard links, this relation is not symmetric: Generally f is not a (symbolic) link to l (although this is possible!). Symbolic links can refer to anywhere across different mount points, and even to non-existing files.

c. Suppose you have created a file f, a hard link h to the same file, and a symbolic link s to f. What happens if you rename f to g? Is the file still accessible via the hard link h? How about the symbolic link s?

Solution:
Renaming f to g just changes the filename in the directory entry; the inode representing the file does not change. The file stays accessible via the hard link h, as it still refers to the same inode as f did and now g does. The symbolic link however breaks, as it referred to the file by its name f, which is now no longer valid.

d. Would the same be true if you had copied f to g first and then removed f?

Solution:
No. In this case, s would be as broken as before, but h would also point to a different file than g. The contents of h and g would be identical, but any change to g or h would no longer be visible via the other name; g and h refer to different inodes, which in turn refer to different (copies of the) data blocks.

e. What happens if you now create a new file f?

Solution:
g and h remain unaffected, s now points to the new file.

f. How can directories be implemented (in the OS)? What information is stored in them?

Solution:
Directories can be implemented as regular files whose type is "directory" rather than "regular file" and whose structure is well-known to the filesystem: It is a list of variable-length records, consisting at least of the filename and the number of the inode that represents the file.

g. Which of the following data are typically stored in an inode: (a) filename, (b) name of containing directory, (c) file size, (d) file type, (e) number of symbolic links to the file, (f) name/location of symbolic links to the file, (g) number of hard links to the file, (h) name/location of hard links to the file, (i) access rights, (j) timestamps (last access, last modification), (k) file contents, (l) list of blocks occupied by the file?

For each item state whether it is required or optional. For items not stored in inodes, state where the information is stored (if at all).

Solution:
(a) The filename is not stored in the inode, as it is not describing the data but merely names it—there can even be multiple names for a single file (see hardlinks)! Filenames are stored in directories.

(b) For similar reasons, no hint as to which directories might contain the file is stored in the inode; each name for the file is stored in a directory which also stores a "pointer" (reference to the inode) to the parent directory, though it does not include its own name.

(c) The file size in bytes is stored ***in the inode****; it is* ***required*** *to correctly deal with files that are no exact multiple of the block size in length.*

(d) The file type is stored ***in the inode*** *and is* ***required*** *to traverse the directory hierarchy: The filesystem at least needs to be able to discern directory nodes (whose inner structure is known and interpreted by the filesystem) from symbolic links (whose content is also interpreted by the filesystem, if supported) and from regular data files. Other common file types exist for block/character devices, named pipes, or Unix domain sockets.*

(e–f) Neither the number nor the identities of symbolic links to a file are recorded anywhere; in the unlikely case that this information is requested, the whole directory tree must be traversed and scanned for matching symbolic links.

(g) The number of hard links to a file is recorded ***in the inode****; this is* ***required*** *to determine when to release the blocks occupied by the file: As soon as the last hard link, i.e., reference to the inode, is deleted, the file itself (inode and data blocks) can/should be deleted/marked as free.*

(h) The identity of the hard links, i.e., their names or locations in the filesystem, is not recorded; checking the consistency of the reference counter requires traversing the whole filesystem and scanning all directory entries for ones matching the given inode.

(i) Access rights are stored ***in the inode*** *as they are considered to regulate access to the data itself rather than to restrict the use of individual names for the file (that's why these rights are not stored in the directory entries!). Depending on various system requirements, access rights themselves can be optional or be implemented in separate data structures (e.g., access control lists), so that this field should be considered optional.*

(j) Timestamps are stored ***in the inode****; but are solely informational: The filesystem does not interpret them in any way. Timestamps are optional but useful for applications such as `make`.*

(k) File contents is generally not stored in the inode; the inode only contains meta-data about the file data. File content is stored in data blocks, whose identity (i.e., number) is stored in the inode (see (l)).

(l) Inodes contain an ordered list of block numbers; each of the blocks referred to in that way by an inode stores a part of the file data.

Question 9.2: Accessing Files and Directories

a. What is the difference between an absolute and a relative path name?

Solution:
An absolute path name starts at the root of the directory structure, whereas a relative path name defines a path starting from the current directory. The current directory is normally process-specific and called working directory.

b. What are the basic methods for accessing a file?

Solution:

Sequential access: *The file is accessed in order, one record (or byte) after the other. A read-operation reads the next n records (or bytes), and advances the position within the file accordingly. Similarly, writes are appended to the end of the file, and the position is set to the end of the file afterwards.*

Direct access: *Programs are allowed to read and write records of the file in any order. In contrast to sequential access, the programmer now has to specify explicitly which record to read/write.*

c. In Linux and Windows, random access on files is implemented via a special system call that moves the "current position pointer" associated with a file to a given position in the file. What are the names of these system calls in Windows and Linux?

Solution:
In Linux, the system call is called `lseek`;
in Windows, `SetFilePointer` and `SetFilePointerEx` are provided.

All functions allow the file offset to be set beyond the end of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Linux:

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

`off_t`, being a 64 bit signed integer, sets the current position in the file `fildes` to `offset`, relative to the location specified by `whence`:

SEEK_SET *The pointer is set to the `offset`'s byte.*

SEEK_CUR *The pointer is set to its current location plus `offset` bytes.*

SEEK_END *The pointer is set to the size of the file plus `offset` bytes.*

Windows:

```
        BOOL SetFilePointerEx(
            HANDLE hFile,
            LARGE_INTEGER liDistanceToMove,
            PLARGE_INTEGER lpNewFilePointer,
            DWORD dwMoveMethod )
```

hFile *Handle to the file whose file pointer is to be moved.*

liDistanceToMove *The number of bytes to move the file pointer. A positive value moves the pointer forward in the file and a negative value moves the file pointer backward.*

lpNewFilePointer *A pointer to a variable to receive the new file pointer. If this parameter is `NULL`, the new file pointer is not returned.*

dwMoveMethod *Starting point for the file pointer move:*

- **FILE_BEGIN** *The starting point is the beginning of the file.*
- **FILE_CURRENT** *The starting point is the current value of the file pointer.*
- **FILE_END** *The starting point is the current end-of-file position.*

d. You can use the `seek` system call to create sparse files: If you seek to a position beyond the end of a file (and if the file system implementation supports sparse files), then the skipped part may remain as a 'hole' without allocated disk storage. Verify this with a short sample program that creates a new file, seeks the position of 1 GiB after the start, writes a single byte, closes the file, and then checks the file size (total size, and block size on disk) using the stat system call.

Solution:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>

int main() {
        // Note: More error checking should be done here after each system call!!!
        int fd = open("test.bin", O_WRONLY|O_TRUNC|O_CREAT, 0644);
        lseek(fd, 1024*1024*1024, SEEK_SET);
        write(fd, "x", 1);
        close(fd);

        struct stat s;
        stat("test.bin", &s);
        printf("File size: %ld\n", s.st_size);
        printf("File size on disk: %ld\n", (off_t)s.st_blocks * 512);
}
```

e. Discuss alternative random access implementations without such a system call.

Solution:
One way is to add another parameter to the `read` and `write` system calls, which supplies the offset to access. This would effectively prepend a seek operation to every `read`/`write` system call and, for truly randomly accessed files, save the extra system call required for `lseek`.

On the downside, this approach adds an extra parameter to every `read`/`write` system call, although most of the time, sequential access is desired. Furthermore, the application programmer or library code must now track the current file position to pass it on to the system calls every time.

Another alternative is mapping the file (or parts thereof) into the virtual address space of the process. This allows for full random access without performing system calls for the accesses at all. Accesses, however, may trigger one or more page faults.

f. What system calls do you need to list the files in a directory in Linux?

Solution:
Before you get a list of file names, you first have to open the directory using the `opendir` system call. You then call `readdir` repeatedly, receiving `dirent` structures that contain information on a single file in the directory or `NULL` when there are no more files. You then call `closedir` to close the used directory stream.

```
#include <stdio.h>
#include <dirent.h>
#include <errno.h>

int main() {
        // TODO: Checking for errors on all system calls
        DIR *dir = opendir(".");
        if(!dir) { perror("opendir"); return 1; }
        struct dirent *dent;
        errno = 0;
        while( (dent = readdir(dir)) != NULL ) {
                printf("%s\n", dent->d_name);
        }
        // if errno==0: all ok (end of directory reached without errir)
        // if errno!=0: readdir failed for some reason
        closedir(dir);
}
```

Note: On Linux, the `readdir` system call has been superseded by the `getdents` system call. The POSIX API `readdir` function is implemented in user mode as a library function that internally calls the `getdents` system call. While `readdir` returns a single directory entry per invocation, `getdents` is able to retrieve multiple entries with a single system call.

Question 9.3: Open Files

a. Discuss the in-kernel data structures that are required to allow for a Unix-like handling of open files.

Solution:
In Unix, the view on open files is local to a process, that is, each process has its own set of open files and thus also its own mapping of file descriptors to files. Whenever a process opens a new file, an entry with a new file descriptor is added to a process-local open file table in the PCB of that process. Additionally, a new element in a system-wide table of open files is allocated, and the PCB-entry in the local table is set to point to that newly allocated global open file element. Each element in the global open file table contains metadata (e.g., the seek position within the associated file and the access rights with which the file was opened) and a reference to a data structure identifying the actual file in the filesystem (a so-called `vNode`). Note that each call to `open` creates a **new** *entry in*

the global open file table, no matter whether or not the same file is already opened. Those entries might point to the same `vNode` data structure, however. This can be seen in Figure 1. Process 1 has opened file A and file B, and process 2 has opened file B. Even though both processes access file B, there are two separate entries in the global open file table. It

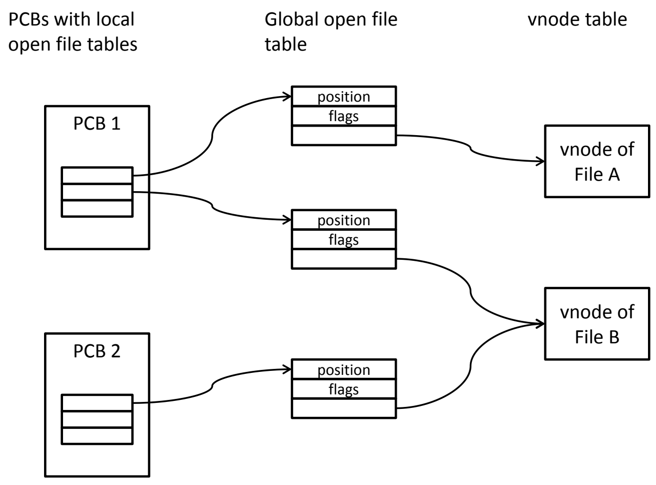

Figure 1: Two processes with open files.

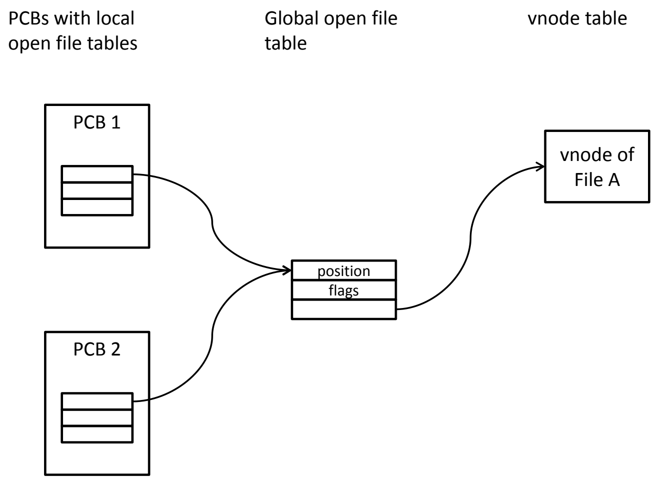

Figure 2: Two processes sharing an entry in the global file table (e.g., after `fork`ing).

can also happen that multiple entries from the PCB file descriptor table point to the same element in the global open file table: If, for example, a process calls `fork`, its PCB (together with the local open file table) is copied. In that case, the child process has its own set of file descriptors that it can manipulate independently from the parent (for example, the child can close a file which still stays open in the parent), but the copied entries in the local table still point to the same element in the global open file table. As a result, a call to `lseek` in the child will also affect the file position in that file in the parent (and vice versa). The situation after executing `fork` is illustrated in Figure 2.