Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 13.1: 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*

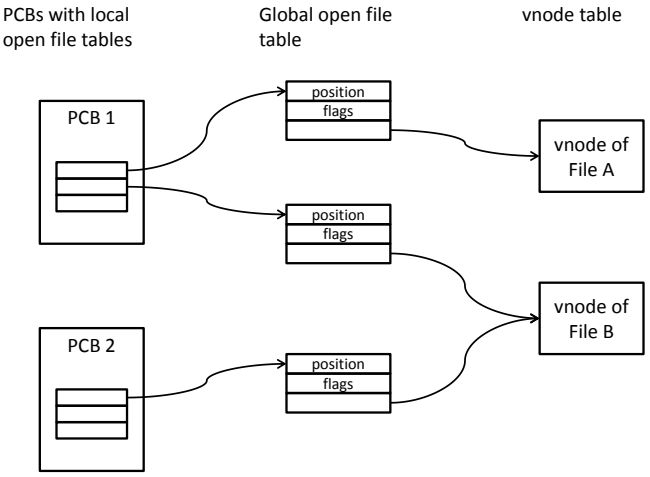

Abbildung 1: Two processes with open files.

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.*

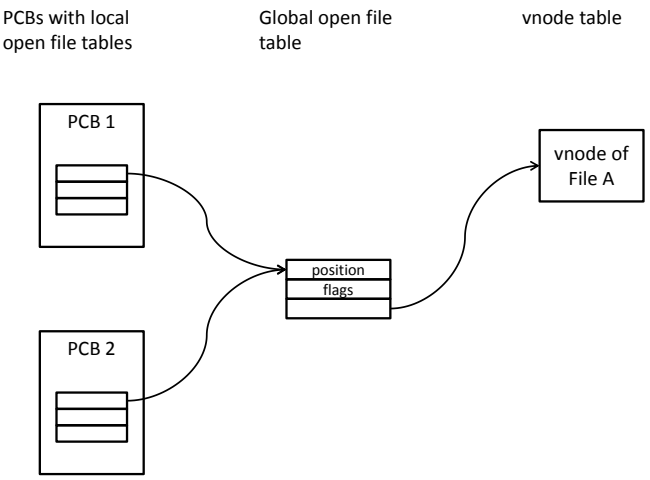

Abbildung 2: Two processes sharing an entry in the global file table (e.g., after `fork`ing).

Question 13.2: Disk Space Allocation

a. How does contiguous allocation work? What are the advantages and problems of this approach?

Solution:
With contiguous allocation, a (contiguous) array of blocks is allocated for each newly created file. A file cannot become larger than the pre-allocated space. The filesystem only needs to store the start block and the size of each file. Contiguous allocation allows for both sequential and direct access: If a `seek` *to position l shall be performed, the corresponding block number can be retrieved by adding $l \div$ blocksize to the starting block. However, contiguous allocation has some severe drawbacks: As files cannot grow beyond the initially reserved size, one has to carefully choose the number of blocks to allocate to each file: If the number is too small, the space for the file will probably not suffice, if it is too large, we will suffer from internal fragmentation. Compaction can be used to reduce external fragmentation, but is rather expensive.*

b. How does linked allocation work? What is its major problem?

Solution:
Linked allocation solves the major problem of contiguous allocation, as files no longer need to occupy a contiguous range of blocks. Similar to contiguous allocation, the file system stores a pointer to the first (and possibly also to the last) block of each file, but each block now contains a pointer to the next block that belongs to the same file. There's no need to pre-allocate disk blocks for files, blocks can be allocated on demand. External fragmentation is no longer a problem. However, linked allocation only allows for sequential access: To find the ith block of a file, the entire list of blocks must be walked. As the blocks reside on the hard disk and **not** *in main memory, walking the list of blocks requires lots of I/O and disk seeks. If a pointer is corrupted for some reason, wrong blocks might be retrieved. Such errors are hard to detect and to correct.*

c. What is the basic idea of a File Allocation Table (FAT)?

Solution:
The FAT-approach is very similar to the linked allocation approach, but stores the list information in a separate data structure (the FAT) (that might be cached in RAM). Still, the entire list has to be walked to find a specific byte offset in a file, but walking the list doesn't require disk accesses if the FAT is cached (even if the FAT is not cached, the amount of

disk seeks is reduced (why?)). Using a FAT still has some disadvantages: The size of the FAT does not depend on the number or size of files, but on the size of the hard disk (there is one entry for each block). For large disks, the FAT might occupy a large amount of disk space, and might also be too large to be completely cached.

d. How does indexed allocation work?

Solution:
Indexed allocation groups the pointers to disk blocks occupied by a specific file together into a single data structure, the index block. There is one index block for each file. Once an index block is loaded to main memory, it is easy to find a specific block of the file: The ith pointer in the index block specifies the location of the ith block of the file.

e. Using indexed allocation, the maximum size of a file depends on the size of the index block. Discuss various approaches that allow for very large files without increasing the size of an index block.

Solution:
The basic idea is to use more than one index block for large files. Different approaches are possible how these blocks are organized: One approach is to construct a chain of index blocks for large files: The first index block contains pointers to n disk blocks, its last entry points to another index block, which again points to n disk blocks and might point to another index block if the number of referenced disk blocks does not suffice. Another approach is similar to multi-level page tables: A first-level index block does not point to disk blocks, but to a number of second-level index blocks, which can point to a third level of blocks, and so on. Entries of blocks of the last level will then point to the actual disk blocks occupied by the file. These two approaches can also be combined: Some entries of a file's index block can point directly to the file's disk blocks (direct blocks), one entry can point to a block of pointers to disk blocks (single indirect block), another entry can point to a block of pointers to blocks with pointers (double indirect block) . . .

f. Consider a filesystem that uses inodes to represent files. Assume that disk blocks are 8 KiB in size and a pointer to a disk block is 4 bytes long. An inode contains 12 pointers to direct blocks, and one pointer to a single, double, and triple indirect block, respectively. What is the maximum size a file can have?

Solution:
*With 4 byte pointers, we can store $8KiB/4bytes = 2Ki = 2048$ pointers in each block. The single indirect block allows to address 2048 blocks, the double indirect block allows to address 2048 * 2048 blocks, and the triple indirect block allows to address 2048 * 2028 * 2048 blocks. In addition, 12 blocks can be accessed directly. The maximum file size can thus be calculated as:*

$$(12 + 2048 + 2048^2 + 2048^3) * 8\textit{KiB} \approx 64\textit{TiB}$$

Question 13.3: 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. They are thus implemented through the file system (not VFS). 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.

b. What are symbolic links?

Solution:
Symbolic links are files of type "symbolic link". Their content is interpreted by the VFS and consists only of the relative or absolute path to 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!).

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? 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) ordered 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. All data blocks except for the last one are completely filled, so that random access to files is relatively cheap: "offset/blocksize" yields the index of the required data block in the list, "offset mod *blocksize" yields the offset of the requested byte in the block. As inodes are of a fixed size, "arbitrarily large" files require a little trick: The last slots in the inode are used to point not to data blocks directly but to indirect blocks, i.e., blocks that just continue the ordered list of data blocks.*

Question 13.4: Virtual File System

a. What is the purpose of the VFS layer in an operating system?

Solution:
The VFS helps to implement a very abstract file API for user applications, and for kernel subsystems. It hides the differences of the specific file system implementations. Thus any kernel subsystem which needs to use file services can use any of the VFS file systems, whether those files are stored in a ext2fs/ReiserFS/XFS/VFAT on a local disk or RAM disk or via NFS/SMB over the network.

b. Discuss potential drawbacks of using a VFS.

Solution:
A potential drawback to using the VFS is that it may hide special features of the underlying file systems. For example, if a file system supports the ability to tune itself for specific

data accesses, the VFS may not offer an access method to applications to specify these special parameters. However, the VFS may permit the application to directly access the lower level parameters of the specific file system (such as via the Unix `ioctl` *system call), but then the application loses any compatibility with other file systems.*

c. Describe the effect of mounting a filesystem.

Solution:
Mounting makes files from a currently inaccessible filesystem accessible. For this purpose, the mount process assigns a name (in an already accessible filesystem) to the root of the to-be-mounted filesystem tree. In other words, mounting combines the namespaces of two filesystems by including the one into the other, adding a fixed prefix (the mount path) to each name in the mounted namespace.