FILE GENERATIONS

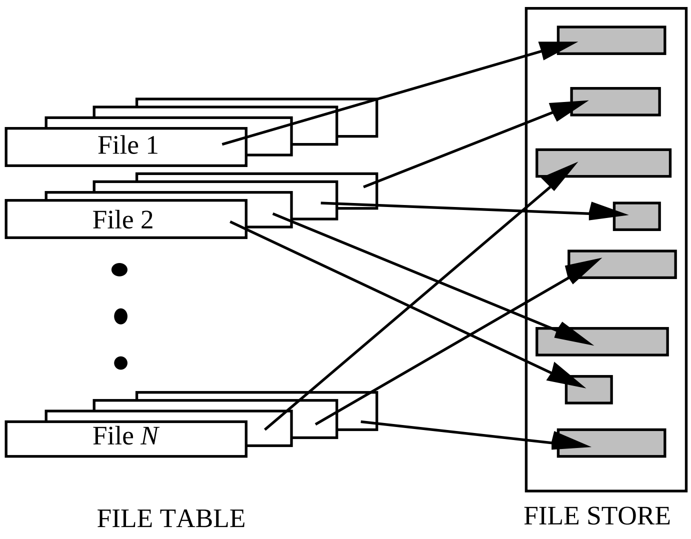
In the previous chapter, we saw that one way to give protection against faults in file operations is to keep many generations of a file. The simplest way to do so is to save multiple copies of the whole file, but that is expensive on space, and commonly inefficient, as many files are only changed slightly at each operation. Nevertheless, the method is commonly used – most commonly as an ad hoc expedient implemented by some programmes, a typical method being to label the generations using filename extensions (.gen1, .gen2, etc.) and then to leave the older generations where they were before. (Extensions .bak1, .bak2, etc. are even more common, but we didn't mention those because we don't believe that these local copies can reasonably be called *backup* copies.)

While this method works in a rough and ready way, it has certain disadvantages. First, as it is not implemented by the operating system, its availability is at the whim of the suppliers of the programmes which might or might not use it, and there is no guarantee that different programmes will use the same conventions. Second, using the file name to store information which is more properly regarded as a file attribute might be necessary if there's no other way to do it, but it's better to find another way. In this case, if the extension is appended to the existing file name, what should happen if the length of resulting name exceeds the limit for the operating system ? – or, on the other hand, if the extension replaces an existing extension, then you can't easily provide satisfactory protection for two files with the same name but different extensions (say, a.x and a.y).

If the facility is useful (and we, like many others, have good personal reasons for believing that it is), then it should be incorporated into the operating system design so that it is available in all circumstances, and always handled consistently. If the requirement is taken into account when the system is designed, necessary facilities can be built into the file directory structures and the system file handling routines. This is an excellent example of the desirability of deciding on the safety features required before constructing the system, for it is much harder to make the requisite changes to an existing file system.

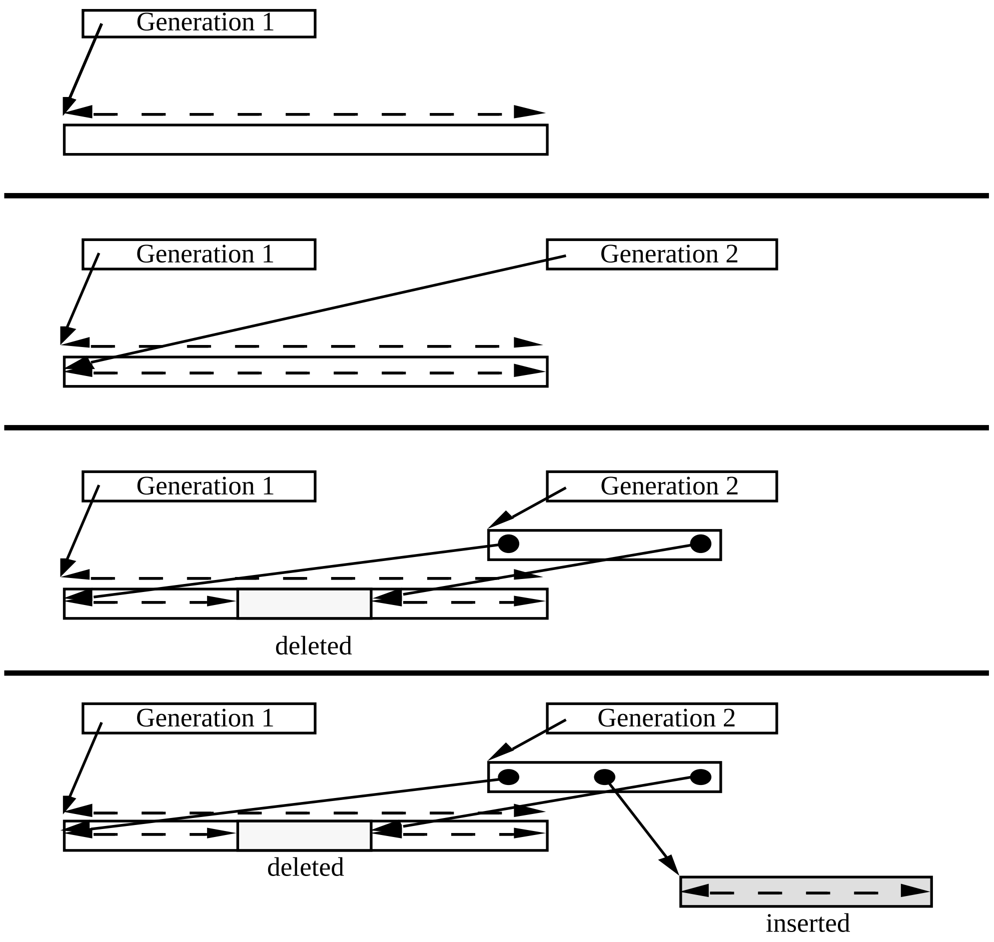
What we require is a system which will keep (in fact, or in effect) multiple copies of the complete file by saving earlier copies each time a new one is made; it could be executed as part of the **open** operation, with provision to revert to the previous state if no change is made to the file. The older copies need not be visible in the file table under ordinary circumstances – recall our remarks on maintaining the system metaphor in the chapter *FILES IN THE SYSTEM* – but can be restored by some variant of an **undo** instruction. (A common name for the operation is **recover**.)

The diagram below illustrates a simple version of one implementation technique. The left-hand side of the diagram shows the necessary (or, at least, desirable) extended directory structure, with several slots for every file name. The front set of entries is the normal file table, while the other entries, if any, are earlier generations of the same file. When one requests a list of files present, one would expect a display of the filenames in the front set; this corresponds exactly to the ordinary list of files. Alternative or modified instructions can be provided to give information about the available older generations.



The arrows indicate that there is just one generation of File 1, and there are four generations of File 2. There are three generations of File *N*, but no front generation; File *N* has been deleted, but older generations will be saved until removed by some "delete all generations" instruction, or until they exceed a set age limit, depending on the discipline enforced by the implementation.

That technique is effective, and comparatively easy to implement, but very expensive in disc space; the cost of changing one bit in the file is the space required to hold a complete copy of the whole file. A more economical techniqueSUP11 gives greater economy by implementing replication at a finer grain, but at the cost of more complex structure inside the file : instead of copying complete files, the changes of structure made within the files as they are modified are recorded. The same sort of "three-dimensional" directory can be used, but in this case each entry points to the root of a tree of pointers, each of which identifies a position on the disc and a segment length. The complete file is obtained by traversing the tree in inorder, and concatenating the segments found. The diagram below illustrates the principle.



The file identified by the first generation pointer is simple, beginning at a defined point and with the length indicated by the dashed arrow. A new generation always begins as a copy of the highest level pointer of the previous generation, but new trees are built where necessary to make changes. The sequence of operations shown constructs Generation 2 by first deleting a segment of the original file and then inserting another in its place. It is clear that, unless the file is largely rewritten at each stage of modification, the cost is significantly less than that of a second copy of the complete file. In one experiment, 14 generations of a file were stored in about the space which would have been needed to store only the first four generations if complete copies had been made.

Notice that both these methods are commonly implemented with all copies on the same medium, because that's the easy way (for the multiple copies) and inevitable (for the tree structures). They are therefore useful for protection against accidental deletions and similar mishaps, but not against catastrophes – and they have no security features at all.

REFERENCE.

SUP11 : F.W. Burton, M.H. Huntbach, J.G. Kollias : "Multiple generation text files using overlapping tree structures", *Computer Journal* **28**, 414 (1985).

QUESTIONS.

What's wrong with the "extensions" method for organising the file generations ?

What are the implications for the file table of saving multiple *generation*s of complete files ?

When a file is deleted, what should happen in both sorts of "*generation*s" system ? Should you ever be able to get rid of a file irreversibly ?