Inter-Office Memorandum

To MBrown, Taft

Date October 8, 1982

From Kolling

Location PARC/ CSL

Subject Internal Functioning of Alpine's FilePageMgr

File <Alpine>Doc>FilePageMgrInternals.tioga

XEROX

Overview

This memo describes the internal functioning of FilePageMgr and supersedes all previous memos on this subject. It is based on memos by myself, Mark Brown, and Ed Taft.

FilePageMgr is a simple file system that is a component of Alpine. It provides files at roughly the same level of abstraction as Pilot does (no access control, no transactions, etc). It sits just above Pilot, in order to isolate the rest of Alpine from Pilot. FilePageMgr is implemented using mainly the Pilot Space and File interfaces; since Space and File are going to change drastically in the moderately near future, FilePageMgr's implementation now is quick and dirty. FilePageMgr depends mainly on the Pilot swapper for data transfers, while retaining the ability to force writes, start write behinds, and request read aheads.

Data access through FilePageMgr is in the mapped style of VMDefs: FilePageMgr is responsible for the assignment of virtual memory to portions of files (unlike Pilot which makes the client responsible for this). The major departure from VMDefs is that FilePageMgr deals with a run of pages called a VMPageSet; there are restrictions on the maximum size of a VMPageSet; as a result, it will be inconvenient for clients of FilePageMgr to manipulate data structures that overlap page boundaries.

General Comments

FilePageMgr is robust. It expects to see and deal couthly with such things as multiple clients trying to access the same page and page requests past the current eof. Other "interesting" concurrent operations will happen without things breaking, although it is expected that the upper levels of Alpine may not actually be requesting those types of concurrency. For example, concurrent SetSizes for a given file will result in both being done, but the order is undetermined. Similarly a page request for a page after the old eof when a file is being concurrently lengthened will either get the page or will get a past eof error, depending on which request gets to the file first.

We have attempted to optimize FilePageMgr's implementation for the client usage that we expect. A client doing "strange things", such as multiple processes touching a file in different modes, will likely experience performance degradation, but data will not be lost.

The basic unit of vm with which FilePageMgr deals is a Chunk, a number of contiguous pages of vm. (The FilePageMgr client deals with a VMPageSet, which corresponds to a Chunk or portion of a Chunk, but which does not give the client access to FilePageMgr implementation details.) There are three main types of Chunks: normal (for file data pages), log (for Alpine's log file data pages), and leader (for file leader pages). Each of these types of Chunks has an associated lru list which is managed by FilePageMgr. If FilePageMgr needs to reclaim vm, it takes the lru Chunk of the appropriate type. The size of each type of Chunk is fixed for a given FilePageMgr instance. An Alpine file has a leader page, which is page 0; we don't ever expect the leader page to take up more than one disk page because additional file properties belong not in the leader page(s) but rather in a distributed database somewhere. FilePageMgr's clients access the leader page in operations separate from operations on the data pages of the file; this is the reason for keeping the Chunks for leader pages separate from the Chunks for non-leader pages, so that we can avoid unnecessary IO.

There is an possible integrity problem concerning the use of multi-page Chunks. When any page of a space is dirtied, the subsequent Pilot write writes out the entire space. This would lead to a problem should a crash occur during the write of a page which has not been dirtied. Because Pilot's uniform swap units currently have performance and implementation problems, we have avoided solving this problem by the use of one page long uniform swap units underlying the Chunks. Instead, each normal Chunk is currently a leaf space and its integrity is preserved by the backup system. Each log Chunk has regular one page swap units underlying it. Each leader Chunk is one page long.

There is currently no way to get Pilot to swap out a multi-page space while guaranteeing that the writes occur in ascending logical page order. The log Chunks were set up with their regular underlying one page swap units so that Mark's design of the log implementation can handle log writes in such a way as to avoid this consistency problem.

FilePageMgr's release routines are called with parameters that tell FilePageMgr whether or not the client is doing sequential or random IO, and if the Chunk has been dirtied. FilePageMgr uses this information to decide where to put the Chunks on its lru lists, and to handle write behind for sequential IO.

Religious statement: if the clients of FilePageMgr respect the pageCounts in the returned VMPageSet, they will never get an address fault.

One sticky point in the implementation is the result of Pilot's restrictions on changing the length of a mapped file. This leads to the following restriction: SetSize returns an Error if there is a Chunk with use count <> 0 which extends beyond either the new or the old eof.

Data Structures and Monitors

A *Chunk* has associated with it at least this information:

fileHandle: needed when taking Chunks off the lru list.
useCount: incremented by 1 for each current user.
startingPageNumber in file.
prev, next links for lru list of Chunks.
info for the red black Chunk tree of the file to which this Chunk is currently mapped.
defWritePending: if this is set, then the Chunk needs to be given to a process for deferred writing. Any Chunk, mapped or unmapped, in an lru list or not, regardless of its useCount may be in the hands of one or more processes for deferred writing. (The unmapped state is due to the chunk being "remapped" and causes no harm.)

In addition to the types of Chunks mentioned previously, Chunks of the following types exist:

treeHeader (for red black Chunk trees of files), lruHeader, and ostGeneral (for use by the red black tree package for its own purposes).

This is how Chunks are allocated and deallocated:

ostGeneral: during initialization, two Chunks of type ostGeneral are permanently allocated for use by the red black tree package.
lruHeader: during initialization, three Chunks of type lruHeader are permanently allocated for use by the lru lists manager.
normal, log, leader: during initialization, a fixed number of Chunks of these types are permanently allocated to FilePageMgr's vm pool.
treeHeader: these Chunks are allocated and deallocated as the system runs. Further details below.

A *FPMFileObject* has associated with it at least this information:

fileHandle: immutable once a FPMFileObject has been created.
chunkTable: red black tree of all the Chunks currently mapped to this file.
nMappedChunks.
fileDataSize: size on the disk, not counting the leader page.
exists: whether or not this file exists on the disk.
nDefWriteChunks: number of Chunks waiting for deferred write.
nLruListChunks: used to protect against a client flooding cache.

A FPMFileObject is allocated when a file is first touched by FilePageMgr (i.e. FilePageMgr is passed a FileMap.Handle such that GetFilePageMgrHandle = NIL). It is deallocated by the garbage collector after FilePageMgr calls FileMap.Unregister to unregister the handle when there are no longer any Chunks mapped to that file. When the FPMFileObject is allocated, the Chunk of type treeHeader for its chunkTable is also allocated; the treeHeader Chunk is deallocated by the garbage collector when the FPMFileObject goes away.

LRU lists

The lru lists of Chunks are maintained by the FilePageMgrLruImpl module. An lru list is a doubly-linked circular list. A list contains the permanently allocated lruHeader Chunk and zero or more other Chunks, the latter all of the same type: normal, leader, or log. There is one such list for each type, making three lists in all. All three lists are protected by the single module monitor, FilePageMgrLruImpl, which is discussed in more detail below. When no process is inside of any FilePageMgr monitor, it is true that a Chunk is on an lru list iff it has useCount = 0. The Chunks on an lru list may be mapped or unmapped; except during state transitions, all unmapped Chunks are on an lru list.

Chunk tables

There is one Chunk table per FPMFileObject. A Chunk table consists of a treeHeader Chunk and a red-black tree of Chunks attached to it. Every Chunk mapped to a file, regardless of the Chunk's useCount, is in the file's chunkTable. Manipulation of Chunk tables is performed through the RedBlackTree package's OrderedSymbolTable interface and protected by its module monitor, RedBlackTreeImpl, which is discussed in more detail below.

Monitors

The FilePageMgr data structures contain a single type of monitor, the FPMFileObject object monitor. One FPMFileObject object monitor is built into each FPMFileObject. In addition, the FilePageMgr implementation contains the module monitor FilePageMgrLruImpl. Monitors can only be nested in this order:

FPMFileObject monitor
RedBlackTree package’s RedBlackTreeImpl
FilePageMgrLruImpl monitor
FileMap’s FileObject monitor

No process ever holds more than one FPMFileObject monitor at a time. A process in a monitor in this list does not necessarily hold all of the "ancestor" monitors in the list.

The various monitors control read and write access to the fields of FPMFileObjects and Chunks as follows:

FPMFileObject:

(FPMFileObject) fileHandle.
(FPMFileObject and RedBlackTreeImpl) chunkTable.
(FPMFileObject) nMappedChunks.
(FPMFileObject) fileDataSize.
(FPMFileObject) exists.
(FPMFileObject) nDefWriteChunks.
(FPMFileObject and FilePageMgrLruImpl) nLruListChunks.

Mapped Chunk:

(FPMFileObject*) fileHandle.
(FPMFileObject*) useCount.
(FPMFileObject*) startPageNumber
(FPMFileObject and FilePageMgrLruImpl, except just FilePageMgrLruImpl for neighbor relinking) prev, next.
(FPMFileObject and RedBlackTreeImpl) rbLLink, rbRLink.
(FPMFileObject and RedBlackTreeImpl) rbColor.
(FPMFileObject) defWritePending.

UnMappedChunk:

Belongs to FilePageMgrLruImpl.

*chunk.fileHandle = nullHandle tells us that a Chunk is unmapped. A process will only write Chunk.fileHandle, Chunk.startPageNumber, and Chunk.useCount when a Chunk is not linked on the lru list (it can tell this because only Chunks with useCounts of 0 are on the lru list); This is because a process in the FilePageMgrLruImpl monitor trying to get a "strange" Chunk wants to know if the Chunk is mapped to a file; if it is not, the process can snarf it up immediately; if it is, it wants to pass back a copy of the fileHandle and startPageNumber as a hint to the caller, who must then get the appropriate FPMFileObject monitor before it can try to free up the Chunk.

The FileMap’s FileObject monitor is used near the beginning of many FilePageMgr operations to protect the checking for and, if necessary, generation of the FilePageMgrHandle field of the FileObject.

Algorithms

The descriptions of algorithms below are generally written in terms of normal Chunks; various portions of the algorithms will be unnecessary for Chunks of other types. The algorithms are also described in terms of each Chunk; in actual fact, we optimize various operations over Chunks. When we refer to mapping a Chunk, it should be understood that that process includes entering the Chunk in the file's red black tree and so forth, with the obvious corresponding operations for unmapping. Uninteresting details of the algorithms and most error handling are omitted.

Most of the procedures initially call GetFilePageMgrHandleAndRegister; this is the routine that checks for the existence of and, if necessary, creates the FilePageMgrHandle under the protection of FileMap's FileObject monitor, and then registers itself as a user of the FileMap.Handle. The corresponding unregistering is done at the end of the FilePageMgr procedure.

ReleaseVMPageSet

Our algorithms assume that Chunks from files declared random access should persist for awhile in the cache (the lru list), while clean Chunks from files declared sequential access are much more expendable. We wish to prevent client files that are declared random access but are really accessed sequentially from flooding the cache. Therefore, if during release of normal Chunks from a file declared random access, an abnormal number of pages from the file are in the lru list relative to the total number of pages in the lru list, we treat such Chunks as sequential. However, because some randomly accessed files may temporarily be legitimately accessed in a sequential manner (for example, when enumeration is taking place), the measures we take to handle cache flooding are transitory.

```
ForceOut the Chunk if this has been requested.
Enter FPMFileObject monitor.
    If this file is normal, find out if it is flooding the cache and, if so, change the Chunk's parameters to sequential.
    Decrement the useCount.
    If the useCount has gone to 0:
        Put the Chunk on the lru list, as mru if dirty or mru requested, else lru.
        If this is a dirty sequential chunk and it isn't already waiting for deferred write, mark it as waiting, increment the count of deferred write waiters for this file, and if that is at its limit, start a process(es) to do some deferred writes. (The Chunks for these processes to handle are found by searching the file's red black tree first backwards and then forwards from the current Chunk position, until we find as many Chunks marked for deferred write as our count of waiters says we should. The defWritePending bit is cleared for each Chunk found.)
Exit FPMFileObject monitor.
```

Demon Process (handles deferred writes)

ForceOut the Chunks we were started with.

Note that when these Chunks were put on the lru list by ReleaseVMPageSet, they were put on as mru, to reduce the possibility that someone looking for a Chunk to remap would get one of them and therefore have to wait for their write. If the caller of ReleaseVMPageSet actually wanted these Chunks to be lru, we do that now; it is done inside the FPMFileObject monitor, and only for Chunks that are still in the same state (same mapping, on lru list, and NOT defWritePending) as when this process started.

ReadPages, ReadLogPages, ReadLeaderPages

Basically, this procedure wants to get a Chunk that is already mapped to the desired state; if no such Chunk exists, the procedure must find another Chunk and map it; of course, the latter may necessitate first unmapping that Chunk from another file, which can only be done under the protection of that other file’s FPMFileObject monitor. Therefore, care is needed to avoid tying up monitors unnecessarily or in such a way as to make deadlocks likely. Two other factors are that we want to order our operations to maximize performance and while we are waiting for monitors, state can change. This is why the procedure works in the following somewhat peculiar way:

```
Set otherChunk to NIL.
DO

Enter FPMFileObject monitor.
    desiredChunk _ NIL.
    Does a Chunk with the desired mapping exist?
        No: if otherChunk # NIL, map it. desiredChunk _ otherChunk.
        Yes: if otherChunk # NIL, put otherChunk on the lru list as lru. desiredChunk _ found
        Chunk.
    IF desiredChunk # NIL, activate it, increment its useCount.
Exit FPMFileObject monitor.
```

If we succeeded, we’re done. Otherwise we’re on the first time thru this loop and we need to get an otherChunk for our next (and final) attempt:

```
DO
Enter the FilePageMgrLruImpl monitor.
    If the lru Chunk is unmapped, remove it from the lru list. If it’s mapped, just remember some
    info about it.
Exit the FilePageMgrLruImpl monitor.
If it’s mapped, we have to enter it’s file’s FPMFileObject monitor and try to unmap it and remove
it from the lru list; if we fail, LOOP.
EXIT;
ENDLOOP.

ENDLOOP.
```

ReadAhead, ReadAheadLogPages

This has constraints on the number of Chunks of the file that it will read ahead. It is basically the *ReadPages, ReadLogPages, ReadLeaderPages* procedure called multiple times followed by *ReleaseVMPageSet* called multiple times with parameters to put the Chunks on the lru list as mru. Unlike vanilla *ReadPages, ReadLogPages, ReadLeaderPages,* it does the Activates of the Chunks all in a bunch.

UsePages, UseLogPages

This is like *ReadPages, ReadLogPages, ReadLeaderPages*, except that it does not Activate the Chunk. In fact, if the client wants the entire Chunk, the Chunk is Killed (even if useCount > 1).

GetSize

```
Enter FPMFileObject monitor.
    Get the size from the FPMFileObject.
Exit FPMFileObject monitor.
```

SetSize

SetSize must beware of Chunks mapped from areas of the file that will cause Pilot distress. For lengthen, it concerns itself with the old eof being "inconveniently mapped." For truncate, it concerns itself with the new eof being "inconveniently mapped," and any Chunks past the new eof being mapped. "Inconveniently mapped" means the page is in the middle of a Chunk; being the last page in a Chunk is not "inconvenient." If these problem Chunks have useCounts # 0, SetSize will return to the client with an error; otherwise, SetSize will write out the "inconveniently mapped" Chunks and Kill the other Chunks, and then unmap them and put them on the lru list. Then it requests Pilot to make the file size change and updates the size field in the FPMFileObject. Naturally this is all done under the protection of the FPMFileObject monitor.

Delete

This is uses the same routines as SetSize; it uses File.Delete instead of File.SetSize.

CreateWithID

Creates the (permanent) file by straightforward calls on Pilot. Initializes the FPMFileObject.

GenerateFileID

Gets an unused unique id from Pilot, so the caller can later supply it to *CreateWithID* .

ShareVMPageSet

```
Enter FPMFileObject monitor.
    Increments the useCount of the Chunk underlying the VMPageSet.
Exit FPMFileObject monitor.
```

ForceOutVMPageSet

Does a ForceOut on the Chunk underlying the VMPageSet. Doesn’t need the protection of the FPMFileObject monitor, since ForceOut on unmapped Chunks is a no-op.

ForceOutFile

```
Enter the FPMFileObject monitor.
    Build a list of Chunks mapped to this file by calling an enumeration routine in the red black
    tree package. The list is constrained in size to avoid this client hogging the system.
Exit the FPMFileObject monitor.
Force out these Chunks.
Repeat, starting the enumeration from where we left off each time, until no more Chunks.
```

ForceOutEverything

Uses the FileMap enumeration routine GetNext (fileHandle) and calls ForceOutFile for each fileHandle.

GetAttributes

```
Enter FPMFileObject monitor.
    Get the attributes by calling Pilot.
Exit FPMFileObject monitor.
```

RestoreCacheToCleanState

This is a routine for debugging. It uses the FileMap enumeration routine GetNext (fileHandle) and tries to unmap all the Chunks for all these files. Then it calls a FilePageMgrLruImpl routine that checks that each lru list contains all the Chunks of the appropriate type and that all those Chunks are unmapped. Clearly, if the system is not quiescent either various errors will be returned or the cache may not actually be clean when the procedure finishes.