A new mount table and protocol to make 9 and Nix faster

Francisco J. Ballesteros

ABSTRACT

It is common to use Plan 9 file servers through slow network connections. Tools like CFS, OP, and others try to help there. In principle, it is the protocol or caching the first target, but the system can be made faster with few changes to 9P and a new mount table. This document describes how.

Protocols

9P relies on RPCs to the server, each requiring a round trip and adding to the total latency for the task at hand.

We have been using the Octopus Protocol, OP [1], to connect to Plan 9 file servers over high-latency network links. But, while this protocol makes it feasible to operate on remote file servers with better performance than 9P, there are still problems. The main one is the lack of compatibility and the need for gateways between 9P and OP.

A second attempt has been IX, a protocol designed to minimize latency. It supports batched requests and streamed replies, including support for conditional retrieval of file data, based on metadata values. Once again, its main problem is the lack of compatiblity.

The kernel

Once a good protocol (for latency) is deployed, the real problem can be seen. The Plan 9 and Nix kernels are designed to issue one RPC for each operation. For example, a *dirstat* requires a *walk* RPC, a *stat* one, and a *clunk* one. In total there are three round trips.

The kernel contains calls to device operations that map to 9P requests in different places, and each one implies a round trip to the server.

For example, consider *pwd*: The shell calls *exec* to run the command, and as a result, the kernel calls *namec* to translate the name *pwd* into a *Chan* structure. A series of requests and replies for *Walk* are issued to locate *pwd* in the current directory and then in */bin*. Because */bin* is a union mount, it is responsible for further requests and replies until *pwd* is located:

```
    Twalk tag 7 fid 275 newfid 273 nwname 1 0:pwd
    Rerror tag 7 ename file does not exist
    Twalk tag 7 fid 170 newfid 273 nwname 1 0:pwd
    Rerror tag 7 ename file does not exist
    Twalk tag 7 fid 166 newfid 273 nwname 1 0:pwd
    Rerror tag 7 ename file does not exist
    Twalk tag 7 fid 90 newfid 273 nwname 1 0:pwd
    Rwalk tag 7 nwqid 1 0:(00000000000001b3 35 )
```

Still within *namec*, the file is opened, because *exec* called with the *Aopen* flag.

```
    Topen tag 7 fid 273 mode 3
    Ropen tag 7 qid (00000000000001b3 35 ) iounit 8192
```

Back in *exec*, the kernel reads the first few bytes to check out which kind of executable it is (binary? interpreted?) and read the executable file header. The file was already in the cache, so the *Chan* is discarded and the cache used.

```
    Tread tag 7 fid 273 offset 0 count 32
    Rread tag 7 count 32 '000001eb 00004ba3 0000095c ...'
    Tclunk tag 7 fid 273
    Rclunk tag 7
```

Now, after executing *pwd*, the program calls the C library function *getwd*, which opens ''.'' and calls *fd2path* to retrieve the path from the kernel's *Chan* for the process current working directory, and the descriptor is closed.

```
    Twalk tag 7 fid 275 newfid 273 nwname 0
    Rwalk tag 7 nwqid 0
    Topen tag 7 fid 273 mode 0
    Ropen tag 7 qid (0000000000002003 139 d) iounit 8192
    Tclunk tag 7 fid 273
    Rclunk tag 7
```

This makes a total of 10 RPCs for printing the current working directory, and by design, each one must wait for the previous one to complete.

Listing a directory is another interesting example. After executing *ls* very much like *pwd* was executed before, the program stats the target to see which file it must list.

```
    Twalk tag 7 fid 274 newfid 275 nwname 0
    Rwalk tag 7 nwqid 0
    Tstat tag 7 fid 275
    Rstat
    Tclunk tag 7 fid 275
    Rclunk tag 7
```

It then opens the target directory,

```
    Twalk tag 7 fid 274 newfid 275 nwname 0
    Rwalk tag 7 nwqid 0
    Topen tag 7 fid 275 mode 0
    Ropen tag 7 qid (0000000000002003 139 d) iounit 8192
```

reads it all to print the entries,

```
    Tread tag 7 fid 275 offset 0 count 8192
    Rread tag 7 count 3373 '3a000000 00000000 80290000...'
    Tread tag 7 fid 275 offset 3373 count 8192
    Rread tag 7 count 0 ''
```

and closes the descriptor when done.

```
    Tclunk tag 7 fid 275
    Rclunk tag 7
```

In total, it required 15 RPCs.

To create a file without writing anything into it we ca redirect standard output at the shell using a null command. This makes the shell issue a create system call, which plays a dance (within *namec*) like shown:

```
    Twalk tag 7 fid 66 newfid 273 nwname 1 0:tmp
    Rwalk tag 7 nwqid 1 0:(0000000000003f73 0 d)
    Twalk tag 7 fid 172 newfid 275 nwname 1 0:x
    Rerror tag 7 ename file does not exist
    Twalk tag 7 fid 172 newfid 275 nwname 0
    Rwalk tag 7 nwqid 0
    Tcreate tag 7 fid 275 name x perm --rw-rw-rw- mode 1
    Rcreate tag 7 qid (000000000000dd06 0 ) iounit 8192
    Tclunk tag 7 fid 273
    Rclunk tag 7
    Tclunk tag 7 fid 275
    Rclunk tag 7
```

It required 6 RPCs.

Is a new protocol enough?

To be able to reduce the number of RPCs it is not enough to have a protocol capable of doing that. The mount table and how the kernel uses files has to change a little bit. On the other hand, it is not enough to have the kernel prepared for the task, the protocol and the server must cooperate.

Plan 9 system calls that take a file name end up calling *namec* within the kernel. This function takes a name and returns a *Chan*, representing a file in use by the kernel (or the processes it runs). Now, to resolve the name, *namec* starts with the process slash or dot file, and tries to walk the given path from there. This requires an RPC to the file server (or device) providing the (slash or dot) file. After having the list of file identifiers (or qids) for the files known by the server for that walk, each element is checked out in the mount table to see if the walking should continue at a different (mounted) file. This means that namec requires:

- Issuing an RPC to the server for the starting point of the file path.
- Looking up the first mount point along the way.
- Repeating this process starting with the first mounted file found (if any), and remaining elements of the path.

When the final file is found, the resulting *Chan* is returned to the kernel routine calling *namec*.

Consider for example the *remove* and *stat* system calls. The former is mostly:

- Call *namec* to get the *Chan* for the file.
- Issue a *remove* RPC to the server providing the file.

The latter looks like:

- Call *namec* to get the *Chan* for the file.
- Issue a *stat* RPC to the server providing the file.

- issue a *clunk* RPC to the server to release the *Chan*.

If the file is after a mounted point, namec would require multiple RPCs, each with a RTT to a server. In other cases (like when opening a file, *namec* has to issue further RPCs, e.g., to open it).

In principle, to remove a file, you could do it in a single RPC RTT to the server, but the mount table, name resolution, and the implementation of remove in the kernel conspire to make you require multiple RPCs (and RTTs).

Other related problems

Long ago Rob Pike published a paper about how to get ‘‘*..*’’ right [3], and the idea was applied to Plan 9. The main problem is that once you cross a mount point, if have to get back, you have to track where you where before crossing the mount point. As an aid, names were added to *Chans* and the syntax of the names was used to try to get rid of ‘‘*..*’’.

Nevertheless, paths starting with ‘‘*..*’’ still remained, and therefore the kernel had to track the mount points crossed while resolving a name, in case it had to walk back.

A related but unnoticed problem is that globbing expressions like ‘‘*../*/*.c*’’ often used to look for files end up walking up just to walk again down, even if some of those are under ‘‘*.*’’.

The plan

The latency problems and the problems caused by dots are around *namec* and the mount table.

First, to simplify things, and avoid the need to keep track of mount points just to walk back, we changed *namec* to do this:

- Clean the path name given to it (nothing new by now).
- If the name is not absolute (i.e., does not start at ‘‘*/*’’), then take the (clean) name from the process dot file and use it to build a clean an absolute path name.

In short, there are no dots in paths from this moment on.

This does not mean that paths are always walked starting at root. If a path lies below the process dot, that file can still be used as an starting point.

As an aside, in the course of implementing this, we found that the clean name routine was not fully correct (regarding paths for kernel devices, for example). That is, after getting ‘‘*..*’’ right it seems it was not fully right. Getting rid of it seems to be a better idea.

With things simplified, the mount table was replaced with another one using names. The table returns the *Chan* for the file that has to be used to resolve a name, considering all files mounted along the path simply as names. Therefore, the first part of the name resolution process does not require reaching a server.

Once this mount table is installed, *namec* is replaced with another routine, called *nameop*, which is given a name and (like *name*) a flag indicating what the kernel wants to do with it. Unlike *namec*, *nameop* has extra values for such flag to indicate things like *stat* and *remove*. In such cases *nameop* takes care of issuing the relevant calls to the device.

This is important because previously there was a call to *namec* and then another to *remove* or to *stat* and *clunk* to release the file. Because there were different calls, the RPCs had to be completed before returning from each call, which means that a better protocol would not be of help to prevent latency problems.

But now, *nameop* can be called, for example, to stat the file with a given name, and it would locate, stat, and clunk it before returning. This is an opportunity to issue all such RPCs concurrently, avoiding separate round trips.

Note that the process is not different of what was done by *namec* to open or to create a file (it did issue those requests). Now it can issue a few more.

The chance to issue concurrent requests does not mean that you can issue them. To do so you require two different things:

- A new mount driver that can issue concurrent requests,
- Different semantics in the protocol than those provided by 9P.

For example, if you want to walk to a file and then to open it you cannot issue both requests concurrently to a 9P server without waiting for each reply after each call. The problem is that the second request might be handled concurrently by the server, and the second one might find that the file identifier (or *fid*) established by the first request does not yet exist.

We took from IX the idea of sequencing groups of operations and created a variant of 9P called 9P2000.ix, or *9Pix*. In this variant requests may use the same tag (concurrently) to indicate that they should be served sequentially (within the same tag value). Also, the semantics of *create* change to mean ‘‘create the file or walk and open it with truncation’’.

Both changes permit the same file server implementation to speak 9P2000 as it would otherwise, but still provide the requirements for issuing concurrent requests and rewriting *namec* and the mount driver to handle them.

Regarding he part of the protocol not described above, i.e., reading and writing files, the new mount driver issues concurrent requests and, for reads, also reads ahead. Therefore, the latency effects are reduced by streaming and by concurrency.

Finally, the mount driver cache is changed to cache also directories and the position where the end of file was found for the files cached. This two changes save a few more RPCs that would be required otherwise.

In what follows we describe the changes one by one.

File names

Each *Chan* has a now a *Path* structure (different from the Plan 9 one) defined as follows:

```
struct Path
{
    Ref;
    char**  els;    /* path elements; no elements means "/" */
    int naels;  /* elements allocated */
    int     slen;   /* size for string with path */
    int nels;   /* elements used */
    int nres;   /* elements resolved */

    Path* next; /* in free list */

    char* buf;  /* if not nil, free(buf) releases els [0:nbuf) */
    int nbuf;
};
```

Paths are kept parsed once cleaned, to make it easy to operate on path elements. Another important point is that paths know how many elements have been resolved and how many elements are left to be walked. This is is an aid both to the new mount table implementation and to defer the evaluation of paths until we know what has to be done with a file (i.e., to issue such requests concurrently).

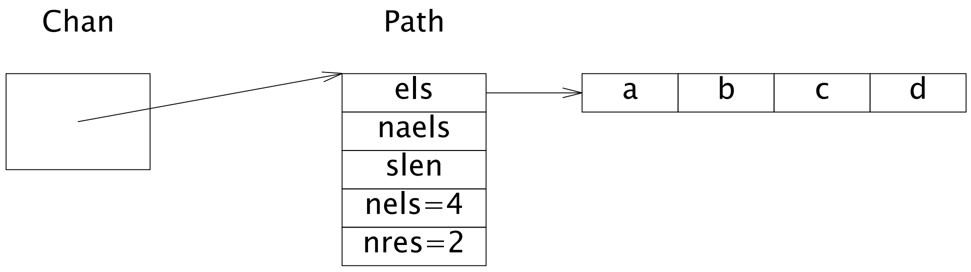

Figure 1 Paths keep parsed names and know how many elements have been resolved. this is for /a/b/c/d with just the first two elements resolved.

To speed up things, *Paths* released are kept in a free list where they can be reused, and they keep the elements array to save calls to the memory allocator.

As an extra aid, the single cleaned string with the path name is split by setting nulls and path elements simply point to bytes within the original name. But, because new elements might be added to a path, we simply note in the *Path* that there is a buffer *buf* responsible for *nbuf* elements in the path.

Most of the times, there is no need to add new elements and the single original string suffices as storage, but when new elements have to be added, *Path* can handled.

As usual, *Paths* are reference counted and can be shared by serveral *Chan* structures. When a path is modified, the routine modifying the path takes care of allocating a new one if required.

Another useful feature is that a new *print* format character, ‘‘*N*’’ has been defined to print paths. It takes care of producing a clean name for any path given. To aid in this process, the *Path* structure keeps the length of the string resulting from printing it, to let the format routine know how much to allocate without going through the elements twice.

All the relevant source is kept in the file ‘‘*path.c*’’ in the portable directory.

The mount table.

The new mount table does not map files to files, like the venerable original one. Instead, it maps names to lists of names and files. This is similar to a *trie* structure that keeps at each node the list of suffixes for a given name. One suffix may be another name element, or a *Chan* to a mounted file. Actually, each name keeps a list of suffixes, which may be of any of the two kinds:

```
/*
 * A mount table and a mount point, depending on your focus.
 * The table is protected by up->pgrp->ns, the rwlock is only
 * for the list of entries to sync with union reads.
 */
struct Mount
{
	Ref;
	Mount*	parent;		/* parent prefix. null if "/" */
	char*	elem;		/* name prefix */
	int ismount;		/* number of Chans mounted in um */

	RWlock;
	Mounted*	um;		/* union mount */
};
```

Each element in the union is defined by this structure:

```
/*
 * Element in union or child mount entry.
 */
struct Mounted
{
	int flags;		/* MCREATE|MCACHE */
	Mounted*next;	/* in union */
	Chan*	to;		/* file in server OR ... */
	Mount * child;	/* ... children mount entry */
};
```

To resolve a path, the trie is navigated to locate the longest prefix that is a mount point for that path. Such longest prefix refers usually to a single *Chan*, which is used as a starting point to walk the rest of the path.

In other cases, the mount point refers to multiple entries (eg., for ‘‘*/bin*’’) and the first *Chan* found is used, although the mount point is noted in the *Chan* to process later the union if needed.

Once the relevant *Chan* is selected, the mount table lookup walks elements of the path yet to be resolved. That is, the mount table maps from a name to an already resolved *Chan* for that name.

It is interesting to note that only a single server is usually required to reach a file, even though there may be multiple mount points along the way.

Permissions are checked on the prefix while the mount point is established, but later, they are just names and carry no mode bits to check. But this no different from setting the current directory at a place where you could walk, because permissions up to the root of the tree are not checked.

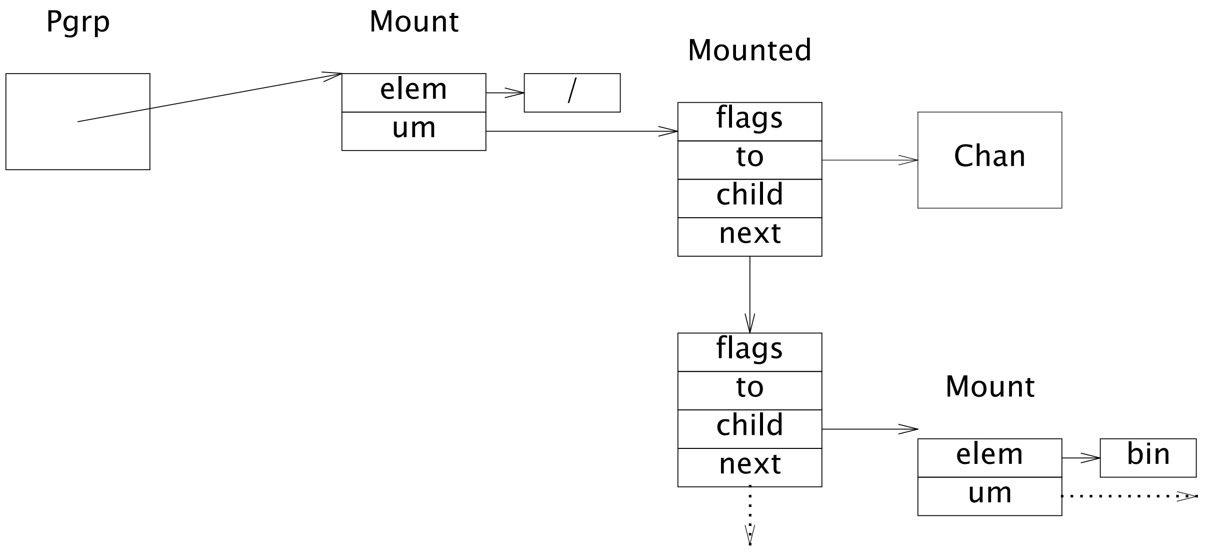

Figure 2 An example mount table. The root has two mounted entries, one to a file in the server and another for a suffix for /bin.

The mount table takes care also of resolving paths that fall under the process dot file or under a device root directory. If the mount table notices that the name for dot is a prefix for the path, and that there are no mounted files after dot, then it relies on the process dot file to continue walking. Otherwise, the mount table is used (ignoring the process dot) both for simplicity (c.f. union directories) and safety.

Paths for devices are always resolved by attaching to the relevant device and walking from there, which means that you cannot pervert a device path by mounting something on it. You can still mount files under ''*/dev*'' or whatever path in the name space is desired, but not under device trees.

In the old mount table there was no difference between a file reached through a name and a file reached through another, but now we can use the names to keep devices undisturbed.

The semantics of the new mount table are close to the old one. But there are a few differences. First, when the name space is cleared using *rfork* the new mount table has to maintain a single entry for *slash* because otherwise no name would exits (but for kernel device names). The old mount table could be cleared because the process *slash* file was used directly for absolute paths.

Second, *binds* that bind a file onto itself are ignored. The old table required this mostly to change flag values and/or to initialize unions. This is no longer required.

Third, the old table defined jumps from files to files, which means that in the sequence

```
bind /a /b
bind /x /a/c
```

the second bind would change what is seen at the target of the first. In the new table names are bound to files. Therefore, the second bind would not affect the result of the first. When an existing file is bound into another, the entries for the former (several if it is a union) are copied into the latter.

Files looked up in the mount table are already resolved, i.e., they have ‘‘crossed’’ the mount point, if any. Therefore, once a *Chan* is obtained, there is no way back. An implication is that the process *dot* has to be resolved again every time the table is changed, or relative paths might not see the effect. This is the result of decoupling the mount table from the *Chans* retrieved from it.

When a name for a union is resolved, the resulting *Chan* is linked to the *Mount* structure for the union. This is used to read union directories by trying to read from each entry in the mount list. But for this, the *Chan* is bound to where it was resolved. That is, any walk performed on it would walk on that file and would not try all the elements in the union. This is not a problem because the mount table resolves that part of the walk before returning, and the current directory is handled as described above. In the case when the current directory is exactly at a union, it is not used to resolve any path. Instead, the mount table is looked up as usual.

The result from these design is that the mount table and the rest of the kernel can be kept mostly decoupled. Nobody else has to actually know how to resolve names in the name space.

The implementation is kept at the ‘‘*mnt.c*’’ source file in the portable directory.

Namec and Nameop

The routine *namec* has been replaced by (and is now a call to) a more general one: *nameop*. This one receives a name and an access mode plus a pointer to arguments for the operation that the kernel wants to perform on the file. Such arguments are similar to the *namec* open mode and permissions, but are able to carry data such as a *stat* buffer. A few new access modes have been defined: *Aremove* is used to remove a file, *Awstat* is used to *wstat* the named file, and *Astat* is used to *stat* it.

The implementation of the *stat*, *wstat*, and *remove* system calls is now a direct call to *nameop* with no extra processing.

In short, *nameop* is responsible for both looking up the name and performing the initial operations on it indicated by the access mode. Unlike before, the access mode may now imply that the *Chan* for the name is to be *clunked* before returning.

Nameop takes care of cleaning up and removing dots from paths as described before in this paper. Therefore, although still there for compatibility, processing for ‘‘.’’ and ‘‘..’’ could be now removed from the rest of the kernel.

Once a *Chan* is returned by *nameop* (if not released before returning), the kernel is free to use it by issuing calls to the driver implementing it. This means that in most cases the server has to be reached before returning from it, and further calls would imply further accesses to the file server.

Another important change is that for creations, *create* is attempted first and then (if it fails because the file exists), open with truncation is attempted. The previous routine would first try to walk to the file before attempting to create it, which means three rounds instead of two.

Finally, *nameop* always returns a non-shared *Chan* to the caller. The previous routine would sometimes return a *Chan* that could be shared and sometimes not, which made necessary for the caller to be aware and call *cunique* to ensure that it was not shared when needed. Because we now *walk* directly the mounted file to the desired target, it is easy to always return a cloned *Chan* and not a shared one.

Split mount driver requests.

The mount driver has been severely changed. Routines like *mntwalk*, *mntopen*, etc. are now implemented by making two calls. For example:

```
static int
mntwstat(Chan *c, uchar *dp, int n)
{
    Mntrpc *r;

    r = mntwstating(nil, c, dp, n);
    if(waserror()){
        mntabort(r);
        nexterror();
    }
    mntwstated(r);
    poperror();
    mntfree(r);
    return n;
}
```

The *−ing* form of the routine issues the call to the server and returns its *Mntrpc* structure. The *−ed* form completes the RPC and provides the result, if any, or raises the error otherwise. This is similar in spirit to Promises [2], although different in both syntax and semantics.

Instead of *mountrpc* and *mountio* there are now *mountrpcreq* and *mountrpcrep* routines to send a request and receive a reply. Multiplexing during reception is otherwise as it was.

The interface for these routines has be carefully arranged so that chains of requests could be made. The first argument of *−ing* calls receives the *Mntrpc* for the previous request in the chain (or *nil* for the first one). The result of the *−ed* calls returns the next RPC in the chain. An example usage follows.

```
/*
 * walk and remove.
 */
p = c->path;
r0 = mntwalking(c, p->els+p->nres, p->nels - p->nres);
if(waserror()){
    mntabort(r0);
    nexterror();
}
mntclunking(r0, r0->wq->clone, Tremove);

r = mntwalked(r0, &wq);
if(wq == nil || wq->nqid < p->nels-p->nres)
    error(up->errstr);
mntclunked(r);
poperror();
mntfree(r0);
```

The new *mntabort* call can be used to release an entire chain of requests, as seen above. Thus, there is no need to install separate error handlers after each request. In the same way, *mntfree* releases also an entire chain. The separate *mntabort* request is needed because when a split request fails, we might have to wait for replies of requests already issued.

Note also in the example above how the new *Path* structure is used to walk what remains to be walked, by looking at the number of elements and the number of elements already resolved.

Because of the new (internal) interface for issuing RPCs, the *Mntrpc* structure has been updated to link requests in the chain through a new *tagnext* field, and to use a double-linked queue in the list of RPCs found at the *Mnt* structure (representing the mount connection).

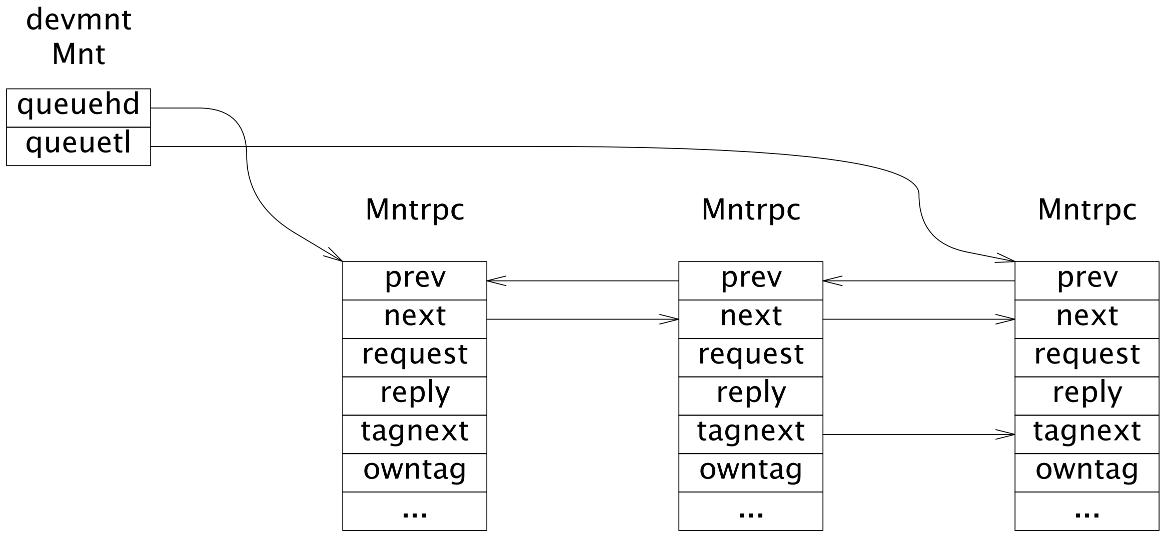

Figure 3 RPCs in the new mount driver. They are double linked in the per mount list of requests and those in the same series are linked through the tagnext field.

Walk processing deserves a note. When *mntwalking* is called a cloned *Chan* (and fid) are allocated in the client. Until the server accepts the new fid, the *Chan* cannot be considered as a mount driver one (because we can't *clunk* a fid that the server has not yet accepted). Once accepted by the server (i.e., once a successful *walk* reply has been received), the *Chan* has to be considered as a mount driver one (because we have to *clunk* the fid before releasing the *Chan* in the client). Also, during errors in the process, the *Chan* has to be deallocated.

To address this, unlike in the previous implementation, the *Mntrpc* contains a pointer to the *Walkqid* structure that holds the cloned *Chan*. Both *mntfree* and *mntabort* take care of releasing such structures if they are found linked from the *Mntrpc* being released. If at some point it is desired to keep them allocated, that pointer can be zeroed. Also, *mntwalking* sets the *Chan* type to zero (so its release is a nop regarding *clunk* operations), and *mountmux* (the routine receiving the reply) sets the type back to the mount driver type when the reply is received. This has to be done in *mountmux* and cannot be done in *mntwalked* because a kill of the process between the request and the reply would leave the fid allocated in the server but not in the client, which means that further requests might fail whenever they re-allocate the *Chan* with the dup fid value.

In the same way, a clunked *Chan* has its type set to zero once we know that the server knows that it has been released.

Because of split requests and replies, flush had to be reworked.

Flushing happens when there is an error during a request (or now, a series of requests) and the kernel wants to be done with that. This is now handled by *abortreqs*.

If the error handled is *Eintr*, which means that the series of requests is being interrupted, a *flush* request is first sent for each different tag found in the list of requests being flushed. During this process, requests issued by other processes must be ignored. In an early implementation this was harder because we used a single list for all requests and not a different one for each process.

Even if several requests share the same tag, a *flush* is sent for each one. The reason is that there might be outstanding requests later in the chain that do not own the tag, and that requests that are found to be completed are ignored at this stage.

Then the replies for the flushes are waited for. Once they arrive, we can deallocate the requests and unlink those not yet done from the outstanding request list.

When the error leading to a call to *abortreqs* is not an interrupt, all outstanding requests must be waited for before raising the error to the user.

9P2000 and 9P2000.ix

It is now feasible to issue concurrent requests in a series, but sometimes they must wait to previous requests in the chain to complete. To address this, a variant of 9P known as 9P2000.ix or *9Pix* has been defined and both *fossil* and the mount driver have been modified to speak it. Both fall down to 9P2000 if the peer is not a *9Pix* speaker. But, when it is, it is guaranteed that requests issued using the same tag value are handled one after another. If one request fails in the same-tag chain, those after it are still processed (and usually they would fail one after another).

The routine issuing RPC requests knows if the connection is *9Pix* capable, and in such case, shares the same tag value used by the previous RPC (passed as its first argument). Otherwise, a new tag is allocated for the new call. This has several implications:

1 The caller must know whether *9Pix* is required or not (i.e., where sequential processing is needed) and issue the RPC chain only if it can be done.

2 The process of allocating and releasing RPCs has now to consider if the RPC owns the tag used or not. In general, for requests sharing a tag value, the first request in the chain owns the tag and is the last one released (to keep the tag value allocated).

Another change in *9Pix* is that *create* is used to create the file (be it by creating it or by opening it with truncation). The *OEXCL* flag can be used by the client (as in the standard system) to ensure that the file did not exist.

Thus, when *nameop* can do so, it calls directly *create* to create the file, instead of trying first to create it and only when that fails trying to point it.

Later may be better

A new device, *later* has been introduced to defer *walks* until we know what to do with a file. Its usage is buried within *nameop* so that nobody in the kernel has to know this device is there.

When a name is resolved and the mount table finds out that the *Chan* in the table can be walked later (i.e., if it is a mount driver channel that speaks *9Pix*) then *walklater* is called to ‘‘walk’’ it instead of the usual call to *clwalk* that would issue the *walk* requests.

This function returns a *Chan* from the *later* device, that has its *Path* structure set to reflect that only part of the path has been resolved, and contains a pointer to the *Chan* that has to be walked later to fully resolve the path.

On those cases that walking later would be a problem (eg., when walking through unions), the *walked* function may be called to ensure that all the path has been resolved in the given *Chan* (or fully resolve it at that point).

Implementation of device operations for the *later* device batch the deferred *walk* with the operation required, and then move the *Chan* back to where it belongs so that it is no longer considered deferred.

For example, this is the code in *Nameop* for the *Awstat* access mode (used by the *wstat* system call):

```
if(nc->ismtpt)
    error(Eismtpt);
arg->nd = devtab[nc->type]->wstat(nc, arg->d, arg->nd);
cclose(nc);
```

For a *Chan* implemented by *later* this call to *wstat* issues concurrent *walk*, *wstat*, and *clunk* RPCs to the server in a single round trip. The *Chan* is then moved out of the device so that the *cclose* call does not actually issue any *clunk* request.

That is, *later* has operations implemented specifically for the usage that *nameop* makes of them, considering that the implementation of *nameop* has to be exactly the same for any other device type.

The example source for split mount driver requests was actually taken from the *remove* operation of the *later* device.

Read and write

It is important to be able to read a file (or to write it) without requiring a round trip for each single read or write. But it is also important to preserve the semantics (regarding coherency) provided by the standard system. Other protocols like *Op* and *NFS* fail exactly at providing the adequate semantics for shared access.

The new mount driver exploits several tools to speed up file I/O, but only for files mounted with the *cache* flag set. Other files might be devices and should be handled with more care.

For files that can be cached:

- Concurrent read or write requests are issued to satisfy each call to read or write the file.
- The cache is used to keep track of where the end of file has been found for each file.
- For (per-process) sequential reads to a file, read ahead requests are issued.

All this adds more pressure on the number of RPC buffers in use, so a limit has been placed on a per *Chan* basis.

The concurrent usage of the protocol described in the previous sections require *9Pix* because of the sequential guarantees, but the tools used and described in this section work also for the standard protocol, because the routine sending requests takes care of allocating different tags if necessary, and because replies are still matched to requests using the tag values.

The new *mntwrite* implementation enters a loop trying to issue requests and waiting for replies only when running out of the per-file limit. Before returning, though, all outstanding requests are waited for. If the user buffer is large enough, a significant speed up may be seen because multiple requests would be issued concurrently. Similar to what the *fcp* program does by hand, but now for everyone using larger buffers. This is the actual code:

```
static long
mntwrite(Chan *c, void *buf, long n, vlong off)
{
    long tot, nw, delta;
    uchar *p;

    if(waserror()){
        abortreqs(c, Twrite);
        nexterror();
    }
    p = buf;
    delta = 0;
    if((c->flag&CCACHE) && c->nio > 0)
        print("mntwrite: %d io0, c->nio);
    if(n == 0){
        poperror();
        return mntrdwr(Twrite, c, buf, n, off);
    }
    for(tot = 0; tot < n; tot += nw){
        nw = sendreq(Twrite, c, p+tot, n-tot, off+tot);
        delta += getreplies(c, Some, nil);
        if(nw <= 0 || up->nnote || delta != 0)
            break;
    }
    delta += getreplies(c, All, nil);
    poperror();
    return tot+delta;
}
```

Here, *sendreq* checks out the limit and issues just the request, waiting for its reply only if above the limit. The call to *getreplies* within the loop scans the *Mntrpc* list to call the receive routine only for those already done, to try to prevent reaching the limit. The call after the loop waits for any outstanding request.

Reading is more complex. The main loop is exactly like that of *mntwrite* but there are a few extra things in *mntread*. Non cacheable files are accessed as it could be expected. For other files, the cache is used to see if there is any cached content. If that is so, that is used (like in the standard system).

Unlike in the standard system, when the cache is exhausted for the file, read ahead is attempted (if the file access for the caller process exhibits sequential accesses). In such case, normal processing is fully replaced. Instead of it, the routine tries to keep a window of read requests issued ahead of the process needs. Note that although the mount driver cache keeps only a prefix for the file, the read ahead processing may be in effect for the entire file, because its implementation does not rely on the cache.

When random access happens, the previously issued requests are waited for (to collect them) and normal processing resumes.

Another interesting bit is that *getreplies* reports if less data than asked for was retrieved from the server, and, in such case, at which offset did that happen. That is used to keep cached the end of file position for cached files. Thus, no extra request has to be issued to re-discover the end of file.

To implement read-ahead, and also to handle the clean up of outstanding read and write requests, the *Chan* structure now keeps a list of *Mntio* entries linked from it.

```
/*
 * There's one of these per chan per process.
 */
struct Mntio
{
	Mntio*	next;	/* in Chan */
	ulong	pid;	/* owner process */
	vlong	loff;	/* last read offset used */
	vlong	raoff;	/* read ahead offset or -1 if not reading ahead */
	int nr;		/* number of I/O requests pending */
	Mntrpc* r;	/* pending read request list */
	Mntrpc* w;	/* pending write request list */
	QLock	wlock;	/* flushed/mntwrite sync */
	IOstats;
};
```

Each process doing concurrent I/O on the mount driver through a *Chan* has one of these linked on the new *Chan.mntio* list. The *r* and *w* lists keep read and write requests. Such requests are linked through the *tagnext* field as shown before while discussing *nameop* but the first request is kept linked at the *Mntio* for the process. The order in which they are linked determines the order in which the replies will be processed by the kernel. Regarding the server, they might be concurrently processed if it is not a *9Pix* server.

The *loff* and *raoff* fiels are used to know what to read ahead and when to get back to random access mode, or to go again into read ahead mode.

It is necessary to keep one such structure per process using the *Chan* to let each different process use its own structure without disturbing others. The *getreplies* routine pays attention to what is wanted and to which process is calling. Another reason for this is that upon errors only the requests for the process involved must be released, but those from other processes should be left alone.

Write behind

Another modification has been made, this time not compatible with the old system. There is a new open mode flag, *OBEHIND*, that permits the kernel not to wait for the reply of the *write* request before returning to the user. The request has been sent before returning, which means that the user buffer is free to use for other things. In this case, a new *fdflush* system call can be used to wait for any outstanding requests and make sure there were no errors during the *write* requests. The per-file limit regarding the number of RPC buffers also applies to write behinds, so the kernel might wait for replies if it runs out of buffers for that *Chan*.

Interesting problems and alternatives

In our first implementation we issued *walk* requests that did not ask for a new *fid* but simply walked an existing one. The requests failed. It seems that *lib9p* does not handle those requests, because the old kernel did not issue them. The protocol supports that feature, but it is too late to use it in the kernel because all existing *lib9p* based servers would fail. It seemed easier to always ask for a clone, which also simplified how the kernel worked because the returned *Chan* is always non-shared.

The handling of the cloned *fids* in *walk* was not obvious at first, and we had to go through a few ‘‘fid already exist’’ diagnostics from our server to discover that there was a race (that was already there in the standard system).

The interface for the device *create* operation is not as it should be. It does not return a new *Chan* (mostly because *clone* files are cloned during open). However, the *later* device needs to replace one *Chan* with another when *create* is called. We had to write an ugly routine that moves the state from one *Chan* to another, so that the newly allocated one was moved into the one passed by the caller. As a result, a new *Chanflds* structure keeps the fiels in *Chan* that has to be cleared, or moved, or copied to aid in routines that have to go over the fields of a *Chan* to copy or zero them.

The initial implementation for read head relied on the mount driver cache to place data read ahead there while waiting for the process to ask for it. This resulted in a slow down. The suspect is the cache because it relies on pages to keep the data cached. Just moving the data out by keeping the requests (and their reply buffers) linked from the *Chan* made the system faster. An interesting side effect is that read ahead is now handled on a per-process basis, thus, programs like *fcp* still exploit read ahead, although the same file is processed with random access when considering all the involved processes.

A drawback of not going through the cache (and not complicating more the implementation) is that files open both for reading and writing cannot use the read ahead mechanism. Otherwise, a data read ahead might be obsolete because of a write performed later, but before the actual call to *read* from the user. During the early testing of the system, only *Acme* was found to exhibit such behavior, for its virtual disk kept on a temporary file.

Adding caching for directories was trivial, only that the cache had to ensure that an integral number of directory entries were returned to the caller, like a server would do.

A previous but finally disregarded idea was adding a flag to indicate if a file was to be used sequentially or randomly. Thanks to Charles Forsyth, we disregarded that. The kernel should know what to do with a file and the user is not to be required to help. Perhaps the write behind open mode flag should also go.

One mistake we made was not placing a per-file limit on the number of buffers. Thus, we did run out of ethernet buffers quickly.

The path handling worked perfectly since the first day, but for names for the *dup* device. In such case, instead of keeping in the *Chan* the path given by the user, *nameop* had to preserve the path of the *Chan* returned. That is so because such device returns another different *Chan* during *open* because of the *dup* performed.

Other protocols like *IX* issues requests in groups, and had to flag the final request on each group. This seemed a good idea but required the server to know when a group was terminated. Thus, a client could stop and the server would keep the group allocated forever. The sequential guarantee of *9Pix* is better because it has only to forward all requests with the same tag to the same worker process in the server, but there is no per-group state left when the tag is (silently) deallocated by the client.

Initial evaluation

Some early evaluation has been performed on the new system. For example, copying *gs* from a far machine took 112 seconds with the old system, and 36 seconds with the new one. Using *fcp* reduced the time further to 19 seconds, because all concurrent processes could still benefit from the changes.

For the typical usage where the system is well connected the new system still seems to go faster.

Enabling just split RPCs and using a 64Kibyte buffer, the old system took 0.0012 seconds to copy a 81K file. The new one takes 0.0006. With caching disabled the old system takes 0.02 seconds and the new one takes 0.0018 (again, using 64K buffers).

This can be explained by looking at excerpts from file server call traces resulting from the new system. For example, this one is from a *dirstat* call:

```
→13 Twalk tag 2 fid 171 newfid 284 nwname 0
→13 Tstat tag 2 fid 284
→13 Tclunk tag 2 fid 284
13← Rwalk tag 2 nwqid 0
13← Rstat tag 2  stat ’tmp’ ’nemo’ ’glenda’ ’nemo’
13← Rclunk tag 2
```

And this other one belongs to actual I/O:

```
→13 Tread tag 11 fid 285 offset 32768 count 8192
→13 Tread tag 12 fid 285 offset 40960 count 8192
→13 Tread tag 12 fid 285 offset 49152 count 8192
→13 Tread tag 12 fid 285 offset 57344 count 8192
→13 Tread tag 12 fid 285 offset 65536 count 8192
→13 Tread tag 13 fid 285 offset 49152 count 8192
13← Rread tag 11 count 8192 ’00000000 02fa0000 01000600 07000800...’
→13 Tread tag 13 fid 285 offset 57344 count 8192
→13 Tread tag 13 fid 285 offset 65536 count 8192
→13 Tread tag 13 fid 285 offset 73728 count 8192
→13 Tread tag 14 fid 285 offset 57344 count 8192
13← Rread tag 12 count 8192 ’892c2489 5c240489 542408e8 fee7ffff...’
```

The kernel issued concurrent reads to keep the read ahead window as the client process was reading. In this case, two different processes were reading ahead the same file. The cache is still updated when data is read ahead, thus, processes that come after the data has been received and processed might still find the data in the cache. Although once that a process exhausts the cache and enters the read ahead discipline, it never goes back to the cache for data, for simplicity.

Acknowledgements

We are grateful to Charles Forsyth for his advice regarding this work, and to the new Plan 9 secret society.

Future work

This work has been done both for Plan 9 and Nix, and we are now in the process of adding these modifications to Nix. We started with Plan 9 because the system is stable and thus we know that any problem is due to the changes made. Nix is still too experimental (like these changes are) to be sure it has nothing to do with the problems.

New operations could be added to the *later* device, for example, to exploit that *exec* wants to read the first few bytes from a file to look at the header. As of now, *nameop* would never call *read*, but adding a new *Aread* access mode to walk, open, and read the first bytes of the file might save some round trip.

References

1. F. J. Ballesteros, G. Guardiola, E. Soriano and S. Lalis, Op: Styx batching for High Latency Links, *IWP9*, 2007.
2. B. Liskov and L. Shrira, Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems, *ACM SIGPLAN Notices*, 1988.
3. R. Pike, Lexical File Names in Plan 9 or Getting Dot-Dot Right, *USENIX ATC*, 2000.