NFS (Client) I/O Architecture & Implementation In Solaris

by
Sameer Seth
Sun Microsystems.

Introduction:

This write-up details the architecture of Solaris I/O implementation for NFSv3 client. The document is helpful in getting insight into the complete life-cycle of NFS data transaction between client and server. The life-cycle involves various steps of NFS data processing by the kernel. These steps within the kernel are:

- receiving NFS read/write request from user application,
- processing data using various kernel framework,
- issuing RPC request to the NFS server,
- (NFS client) receiving the response from the NFS server &
- processing the data and finally returning to the user application which initiated the request.

Both SYNC & ASYNC framework for NFS client data transaction is elaborated. The idea is not to walk through the entire code base but to get familiar with the design and implementation of NFS clients read/write process in the multi-threaded kernel environment. Various kernel data-structures involved in client's NFS read/write is covered. Various kernel framework used by NFS client like kernel VFS, paging, VM, NFS etc., are touched, though not in-depth. This write-up contains certain examples that explain NFS clients read/write behavior in different situations. Clients data and attributes caching is covered explaining in brief open-to-close consistency implementation.

This is helpful in understanding & tackling the read/write/caching and related issues associated with NFS client. Not only this, the document serves as a roadmap for NFS v3 read/write process at the client end. Last but not the least, this write-up doesn't cover each and every details of NFS client related to the subject. So, there is a great scope for anybody interested to add more to this. NFS v4 read/write architecture is not very different from v3 except for delegation feature(serialization of read & write) & compounded RPC calls but they have not changed the NFS read/write architecture and design. The comparative study of NFS v3/v4 can be the next step to strengthen our belief.

Since this is the first document mapping the source & the architecture, there is likely to be some errors. There is going to be errata for all such errors.

NFS (client) read I/O Architecture:

1. if caching has been disabled or if using client-side direct I/O and the file is not mmap'd and there are no cached pages, bypass VM. Fig. 1 describes the NFS v3 read steps.

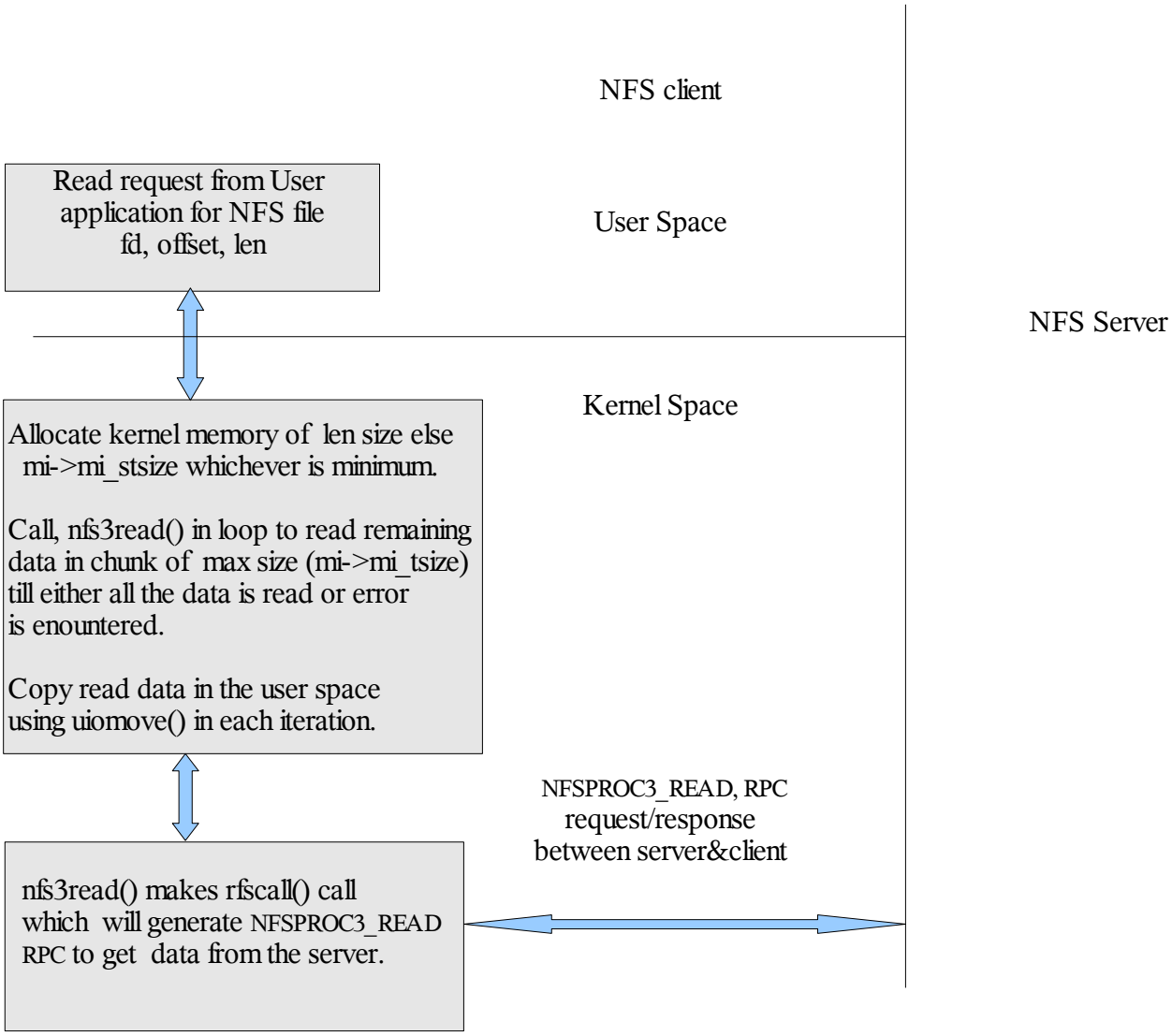
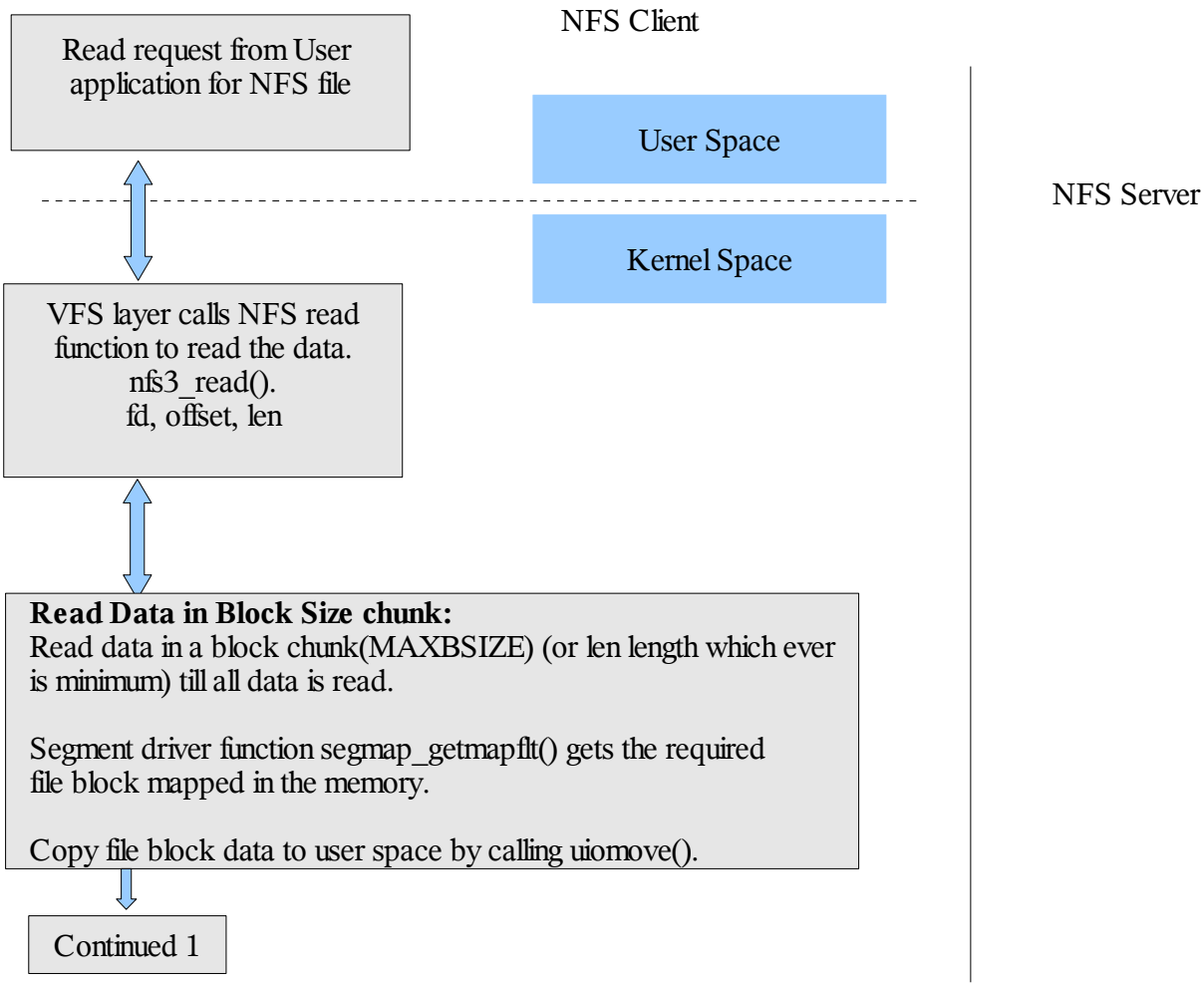
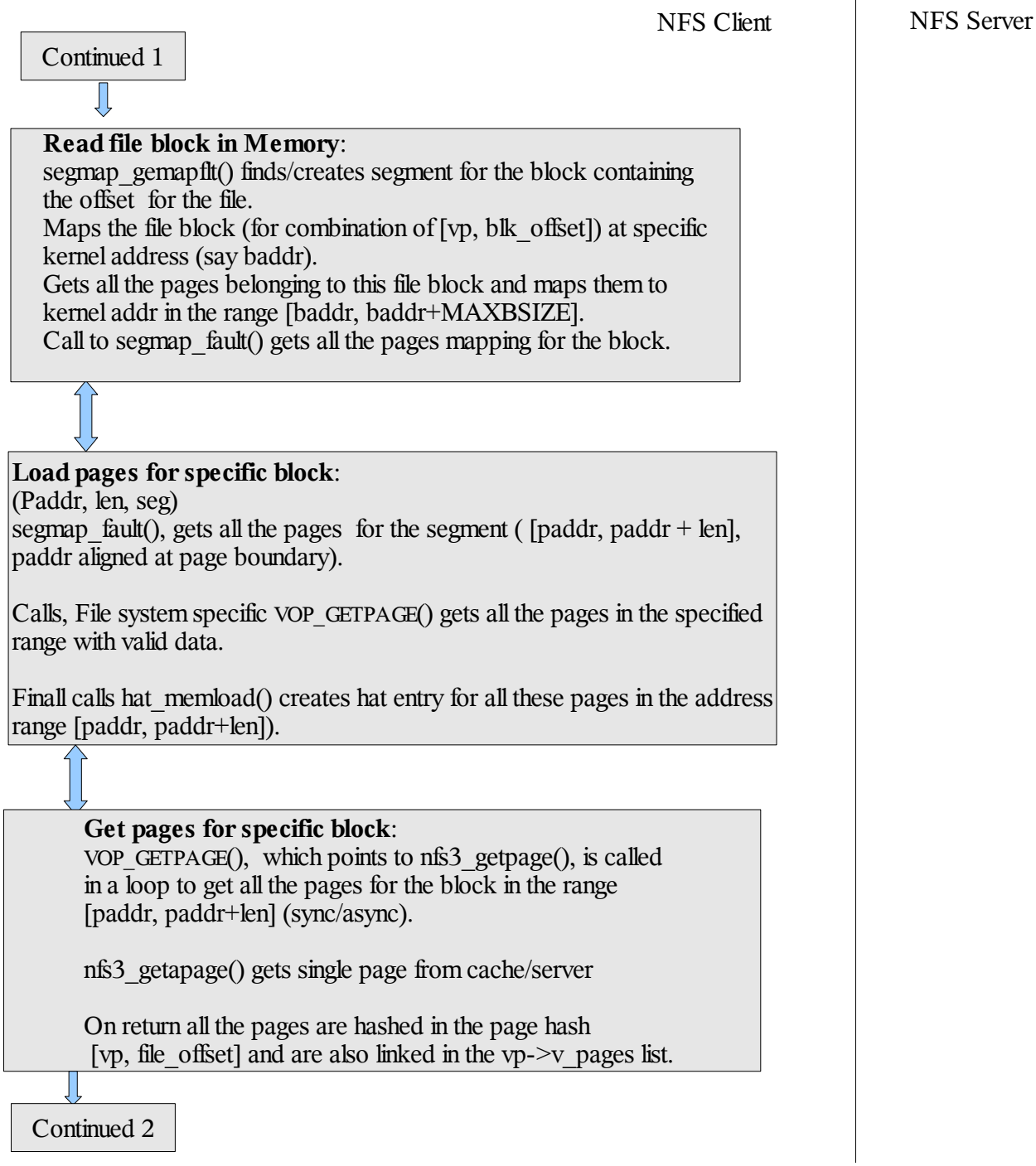


Fig. 1

2. Consider a case where caching is enabled or we are using client-side direct I/O with file mmapped or cached pages. Here we need to go through kernel VM framework to allocate pages to contain the file data. Each kernel page for this vnode will contain data for the combination of [vp, file_offset]. The pages will be mapped to kernel virtual addresses allocated by the segment driver for each block of the file for the combination of [vp, block_off]. We need to cluster the pages around the requested page(within the NFS block boundary) in the final block I/O request. Once we identify all the pages to be read, we map them to the kernel virtual address and place NFS block I/O request to read the portion of the file from the server using RPC call. Finally, data for the requested portion of the file is read in the pages which are mapped to kernel virtual address and we can read the data accessing the address. Fig. 2 explains the NFS v3 data read steps with caching enabled.




NFS Client
NFS Server
Continued 1
Read file block in Memory:
segmap_gemapflt() finds/creates segment for the block containing the offset for the file.
Maps the file block (for combination of [vp, blk_offset]) at specific kernel address (say baddr).
Gets all the pages belonging to this file block and maps them to kernel addr in the range [baddr, baddr+MAXBSIZE].
Call to segmap_fault() gets all the pages mapping for the block.
Load pages for specific block:
(Paddr, len, seg)
segmap_fault(), gets all the pages for the segment ([paddr, paddr + len], paddr aligned at page boundary).
Calls, File system specific VOP_GETPAGE() gets all the pages in the specified range with valid data.
Finall calls hat_memload() creates hat entry for all these pages in the address range [paddr, paddr+len]).
Get pages for specific block:
VOP_GETPAGE(), which points to nfs3_getpage(), is called in a loop to get all the pages for the block in the range [paddr, paddr+len] (sync/async).
nfs3_getapage() gets single page from cache/server
On return all the pages are hashed in the page hash [vp, file_offset] and are also linked in the vp->v_pages list.
Continued 2

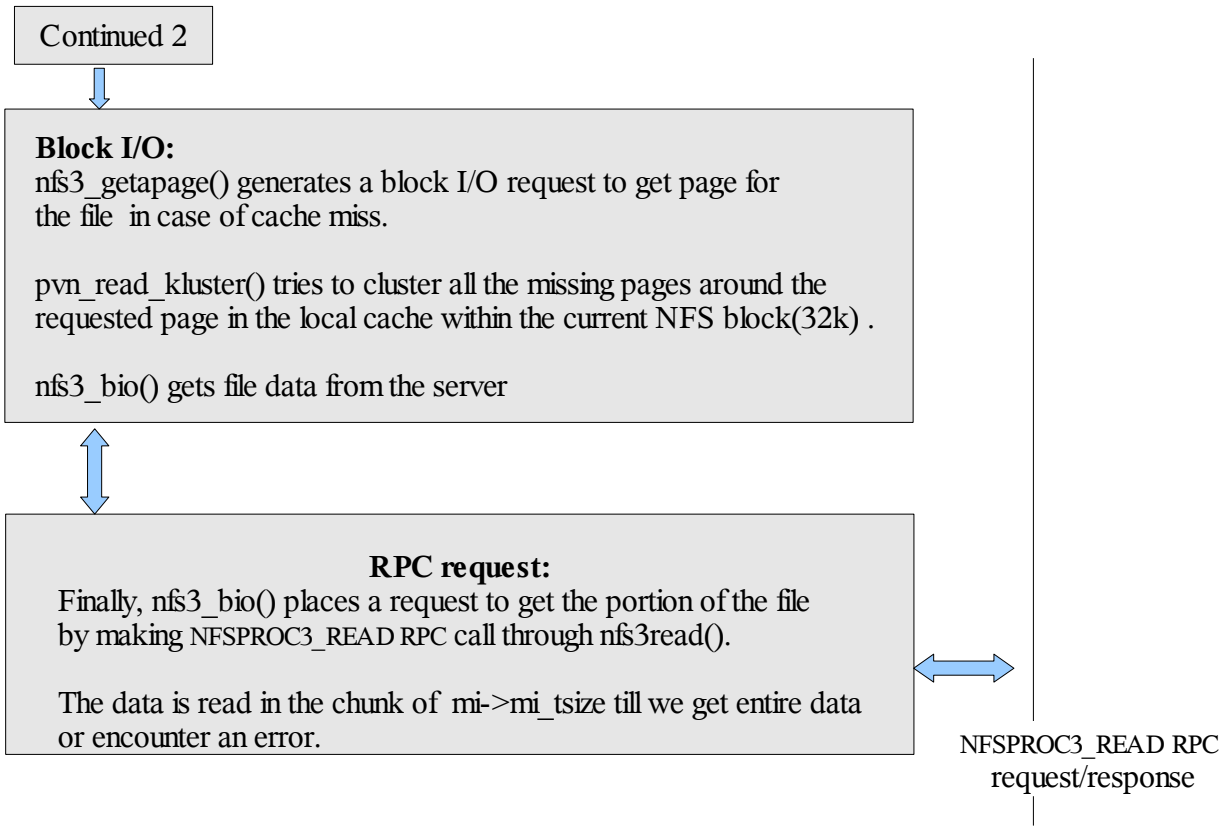

Continued 2
Block I/O:
nfs3_getapage() generates a block I/O request to get page for the file in case of cache miss.
pvn_read_kluster() tries to cluster all the missing pages around the requested page in the local cache within the current NFS block(32k) .
nfs3_bio() gets file data from the server
RPC request:
Finally, nfs3_bio() places a request to get the portion of the file by making NFSPROC3_READ RPC call through nfs3read().
The data is read in the chunk of mi->mi_tsize till we get entire data or encounter an error.
NFSPROC3_READ RPC request/response

Fig. 2

2.1 Mechanism of NFS read with caching enabled

Fig. 3 shows sequence of important function calls to read NFS data.

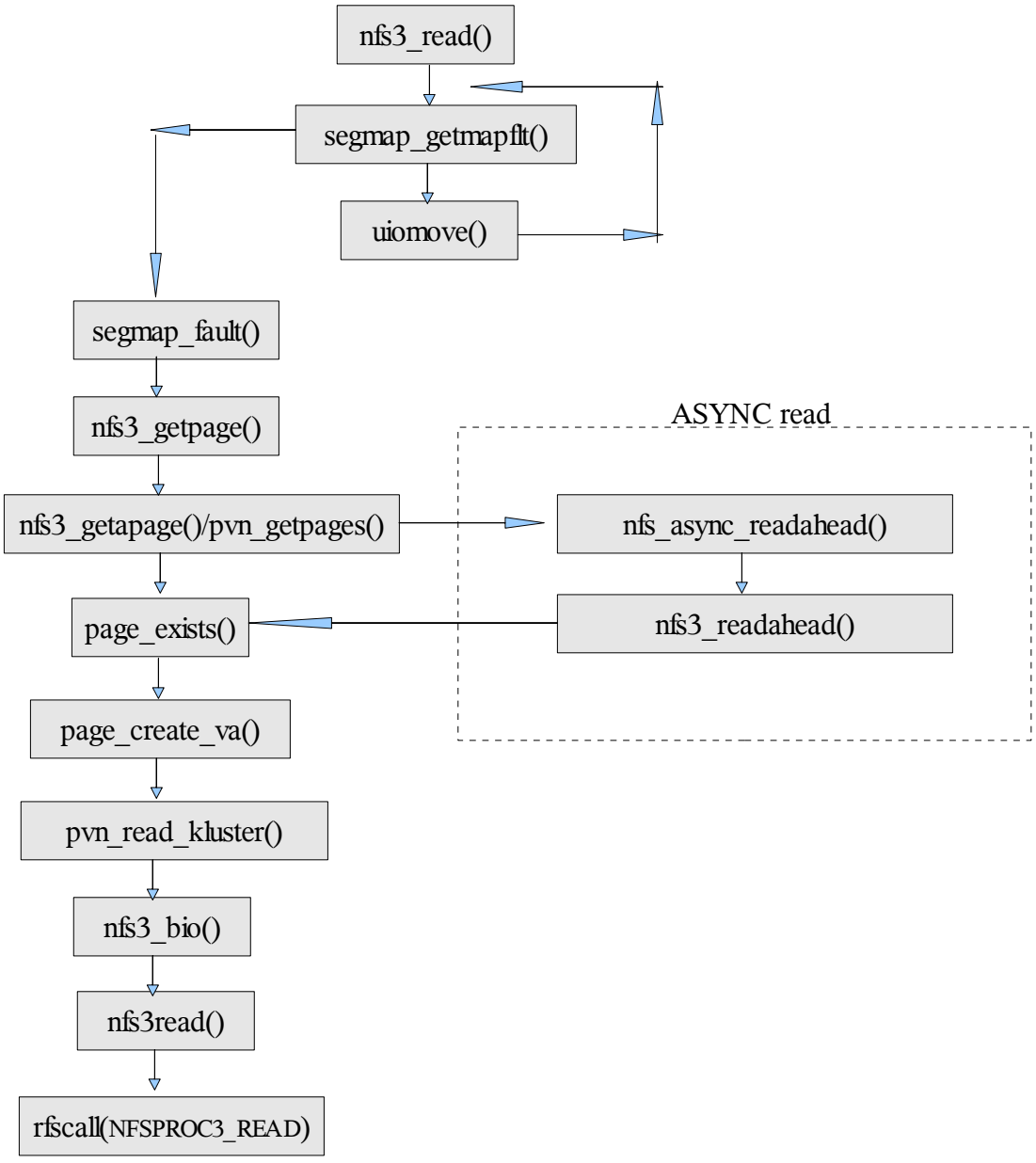


Fig. 3

Page & Block I/O in NFSv3 read operation:

This section explains the steps involved in processing NFS read request(by kernel) from user application when caching is enabled. The entire read process involves various kernel subsystems like VM, paging, block I/O etc.,. We see that read request is handled by different kernel sub-systems. Kernel needs to map the portion of the file to kernel virtual address(segment driver), allocate pages for this portion of the file(pagging), read file data into these pages (block I/O) and finally loads these pages (with valid file data) to the kernel virtual address allocated by segment driver (VM) so that data can be read in by accessing the addresses. In the entire explanation we consider file length (len) to be read = 9000 bytes from file offset (f_offset) = 17000 and where ever required the changed values are mentioned to explain different scenarios.

Application has requested to read 'len' length of data from offset 'f_offset'. One block of data (MAXBSIZE) is to be read at a time. First the file block is identified which contains the file offset 'f_offset' and then the subsequent blocks(depending on length to be read). There is a specific kernel segment mapped to each file block [vp, b_offset]. segmap_getmapflt() is a segment driver function which either gets us already mapped segment for the file block or creates one. It manages kernel virtual addresses of length MAXBSIZE aligned at block boundary for the file blocks. All the pages for file block [b_offset, b_offset+MAXBSIZE] are mapped to file block segment[baddr, baddr+MAXBSIZE].

Lets say MAXBSIZE is 16k and PAGESIZE is 8k in size. For f_offset is 17000, the block offset for the block containing f_offset will be 16384.

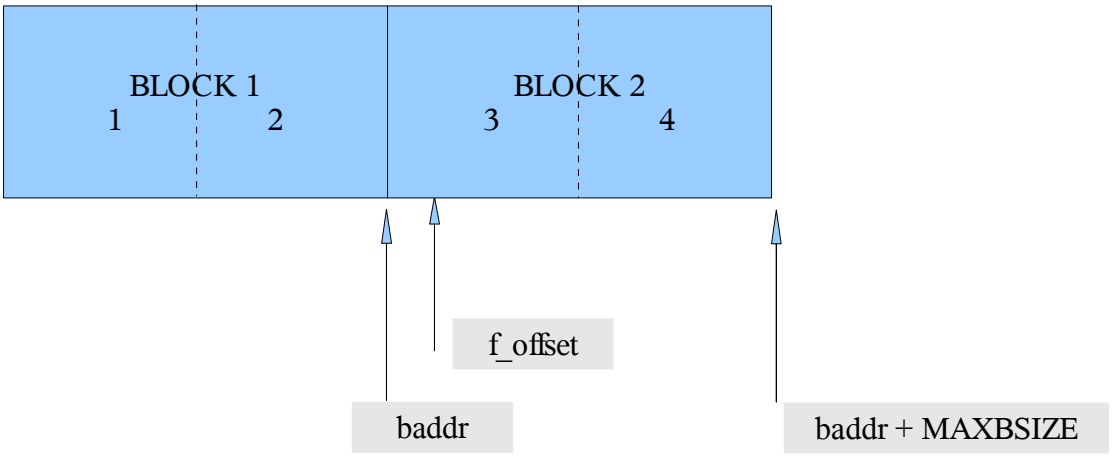


Fig. 4

block 1 contains pages 1 & 2 and block 2 contains pages 3 & 4. Current f_offset lies in page 3 (of block 2) as shown in Fig. 4. segmap_getmapflt() gets the kernel virtual address for block 2 as baddr where this block is mapped. Now we need to get the required pages (with valid file data) for this block and map them to address range [baddr, baddr+ MAXBSIZE]. Number of pages to be read depends on the file length to be read from offset f_offset. For len 9000 Bytes, we need to read page 3 & 4 for block

segmap_fault() is called whenever page fault occurs on a specific address which gets the required pages for the address and maps them to address(where fault occurred). We call this function from segmap_getmapflt() to get the pages 3 & 4 for block 2 and map these pages to address range [baddr, baddr+ MAXBSIZE].

segmap_fault() calls macro VOP_GETPAGE(), which points to nfs3_getpage(), to get the pages. Finally, when all the pages are read in, they are mapped to the address [baddr, baddr+ MAXBSIZE] (hat entry is created for these pages) by segmap_fault(). After segmap_fault() has returned, the pages with valid file data are loaded in the memory at virtual address baddr and can be accessed using this address Fig.5.

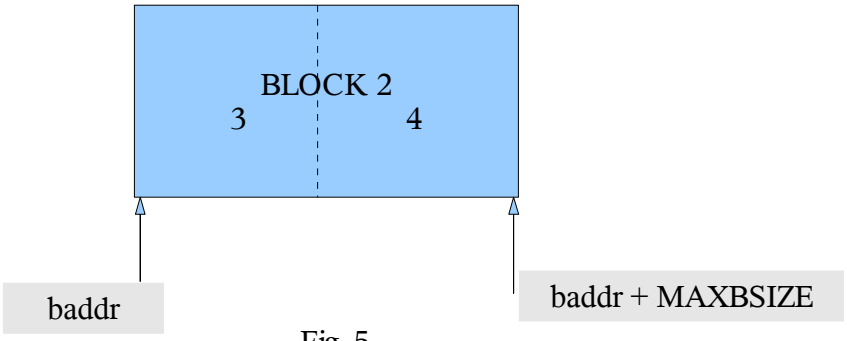


Fig. 5

Let's look at nfs3_getpage(). This function calls nfs3_getapage(), if only single page is to be read else it calls pvn_getapage(). pvn_getapage(), does nothing but calls nfs3_getapage() till it has read all the pages and returns all the pages in an array of page_t structure. This array is returned to segmap_fault(), which maps these pages to the address range [baddr, baddr + MAXBSIZE] in our example as shown in Fig 5.

nfs3_getapage() gets the file offset (aligned at the page boundary) and the file length to be read. It acts on new block size specific to the NFS. The new block size is calculated using following macro –

bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE). <----- maximun of the two values is selected.
boffset = f_offset(aligned at page boundary)/bsize, so new block offset is boffset.

For NFS, vp->v_vfsp->vfs_bsize is 32k. Here onwards the new block size is 32k and all the calculations are based on this block size. nfs3_getapage() checks for the possibility of any read ahead operation (ASYNC operation, explained later) to read ahead blocks of data around the requested block in advance by calling nfs_async_readahead(). If the readahead is done and it is successful, it would have got all the required pages. It looks for the required page in the page hash page_exists(). All the pages (neither freed nor invalid) are hashed in the page hash table for the combination of [vp, offset] where offset is the file offset aligned at page boundary and vp is the vnode for the file. If the page for the file offset f_offset is found in the hash table, acquire the appropriate lock (exclusive lock) by calling page_lookup() and return the page. In case the page is not found in the cache (hash table/free list), we need to get the page from the server. So, we get prepared for the block I/O to read the required pages within the nfs block from the server. First we allocate a page for the current page (containing f_offset) calling page_create_va().

This function allocates a page for [vp, offset], gets exclusive lock & I/O lock on the page adds the page to the page hash table and also to vnode's page list (vnode->v_pages list linked by page->p_vpnext & page->p_vpprev). Find out how much data actually needs to be read. We have client's view of file size rnode->r_size. Refer Fig. 6 for this explanation.

1. If the file size (rnode->r_size) lies within the current NFS block boundary (NFS block which contains the file offset).
2. if f_offset is more than or equal file size (which means that tat we want to read beyond the EOF), we will read atleast one page. (r_len = f_offset + PAGESIZE - boffset)
3. else if the file size is more that the offset, we will read the file length within this NFS block till the EOF (r_len = rnode->r_size – boffset).
4. else if the file size lies outside the current NFS block (boffset), we need to read the entire block if we are not reading the start of the file (r_len = bsize).

Now, it tries to cluster all the pages (which need to be read in) around the required page within the NFS block boundary (within the boundary [boffset, boffset+r_len]) by calling pvn_read_kluster(). pnv_read_kluster() checks for all the pages around the requested page (within the NFS block) if they are to be read and adds all those pages in the page I/O list (linked by page->p_prev, page->p_next pointers) . It calls page_create_va() with the flags set to PG_EXCL | PG_WAIT for all those pages. Which means that if the function returns NULL, the page we are looking for already exists in the cache else we get a newly allocated page with the exclusive lock& I/O held(by the current thread) on the page. Let me explain it with the help of our example. The page we require is page 3 (in block 2). Since, NFS block size is 32k we will scan through all the pages 1, 2 & 4 (around page 3) because they belong to single NFS block. If we find that the page 2 and 4 are missing in the cache, these pages will be allocated and linked in the page I/O list and finally the new file offset and file length are returned to the calling function along with the page I/O list. After return from the function offset and file length for read operation will be as follows -

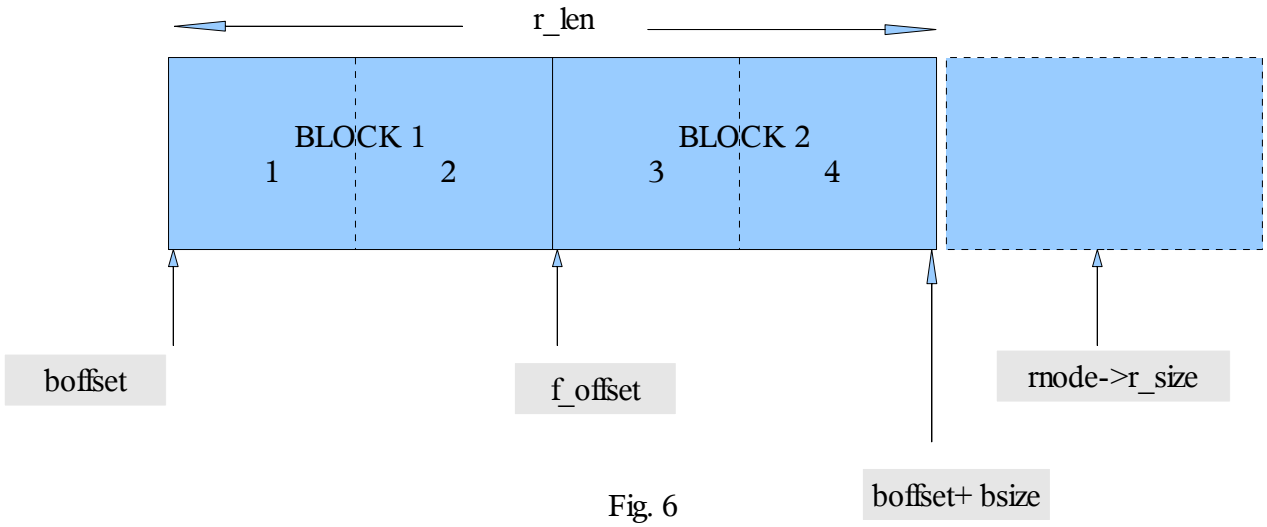


Fig. 6

1. Consider a situation where rnode->r_size (file size) is is beyond 32k and file offset is 17000 (requested page is 3). In this case, r_len will be one NFS block size (32k) which means all the three pages (1, 2 & 4) around page 3 will be scanned by pvn_read_kluster().
2. If file offset (f_offset) is 10000 (requested page is 2) and file size is 25000, r_len will be 3*PAGESIZE. Which means only pages 1 & 3 around page 2 will be scanned.
3. If f_offset is 17000 (requested page is 3) and file size is 16384, r_len will be 2*PAGESIZE. In this case, page 1 & 2 will be scanned around page 3.

In case we are reading beyond EOF and nfs3_getpages() is called as a result of NFS write operation, we zero out the current page and return else if it is read operation, return EOF.

Now, if we need to read data from the server, we need to place a block request to the NFS server to read the required data. In our example, page 1, 2 & 4 are scanned and page 2 & 4 are missing in the cache. So we cluster the block request for three pages 2, 3 & 4 with new file offset as 8k (starting from page 2) and file length as 24k(read data for pages 2, 3 & 4). Initialise buf struct for this request and remap these pages to a new kernel addresses (these pages are not yet mapped as they will be mapped in segmap_fault() on return) by calling bp_mapin(). The portion of the file (offset = 8k, len = 24k) is read by calling nfs3_bio(). nfs3_bio will call nfs3read() which generates NFSPROC3_READ RPC request to read data in the chunk of max mi->mi_tsize size till either all the data is read or error is encountered. When nfs3_bio() returns, we need to unmap the mappings for the pages read in (page 2, 3 & 4 in our case) by calling bp_mapout() and return the required pages to the calling function. Finally segmap_fault() loads the pages(3 & 4) in the memory by mapping it at [baddr, baddr+MAXBSIZE]. This way we have read the required file block (block 2) data from the server which is mapped to a specific address baddr (Fig. 5). File data at offset f_offset can be read using baddr (block base address + offset within the block) and uimoved from kernel to user space.

NFS (client) write I/O Architecture:

1. If caching has been disabled (e.g., locking) or if using client-side direct I/O and the file is not mmap'd and there are no cached pages, bypass VM. In this case, block & page i/o is skipped and file data will be directly written to the server in maximum chunk of mi->mi_stsize chunk till all the data is written back to the server. We write only if rp->r_flags is not set to RDONTWRITE. This means some write error, like quota full, max file size has occurred on the file in the last NFS write operation because of which further write's will not be accepted by the server(Fig. 7).

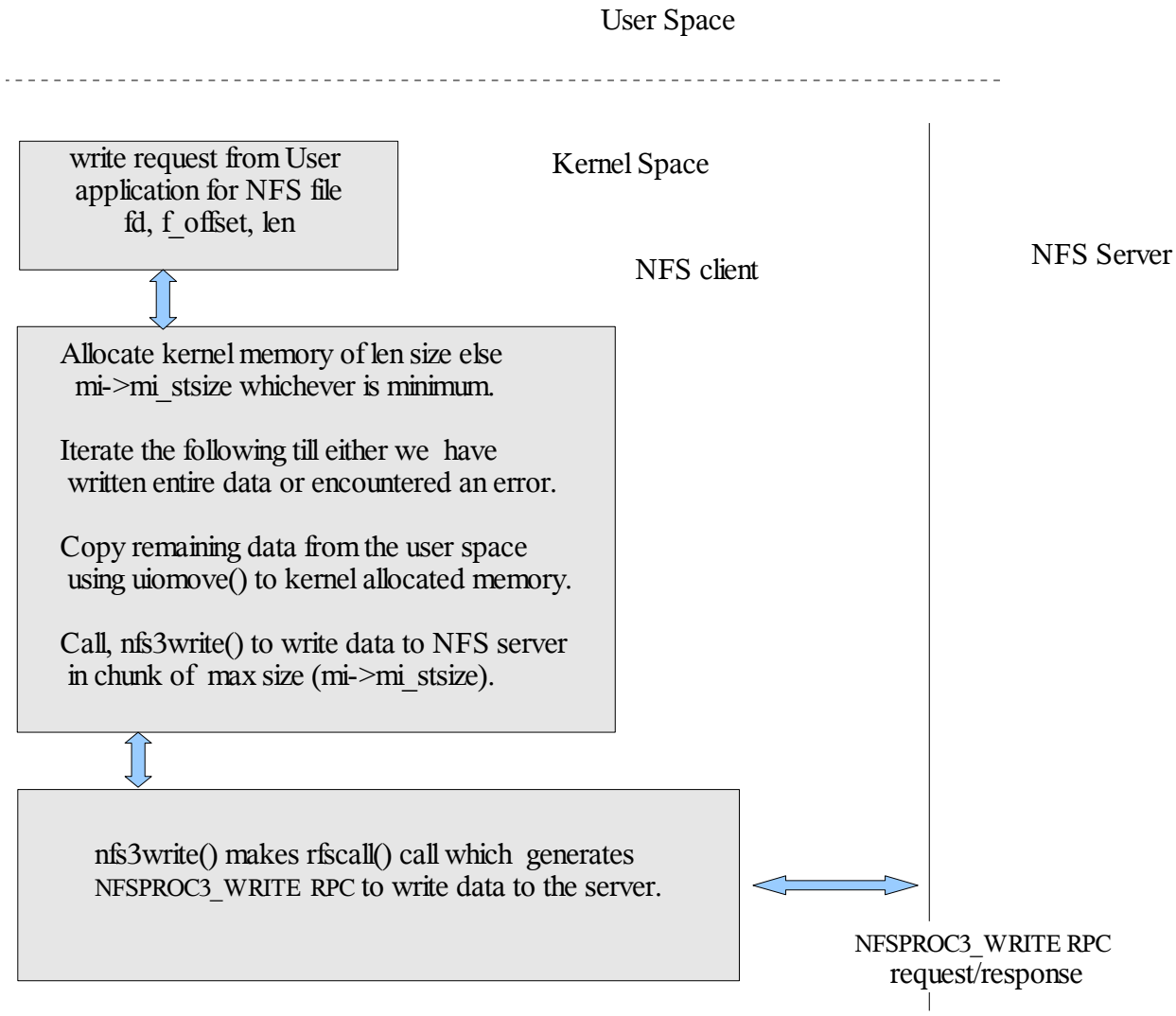
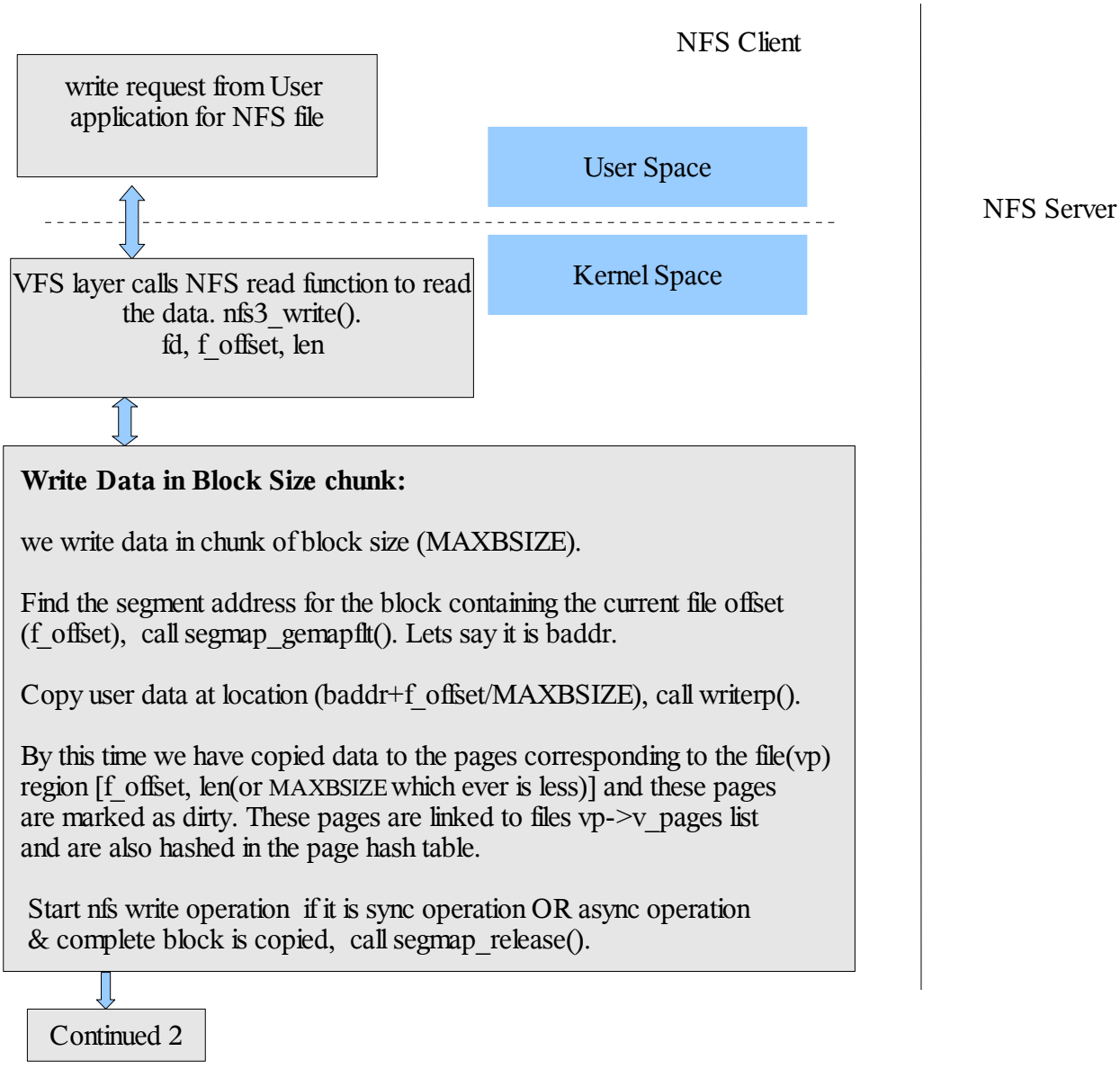
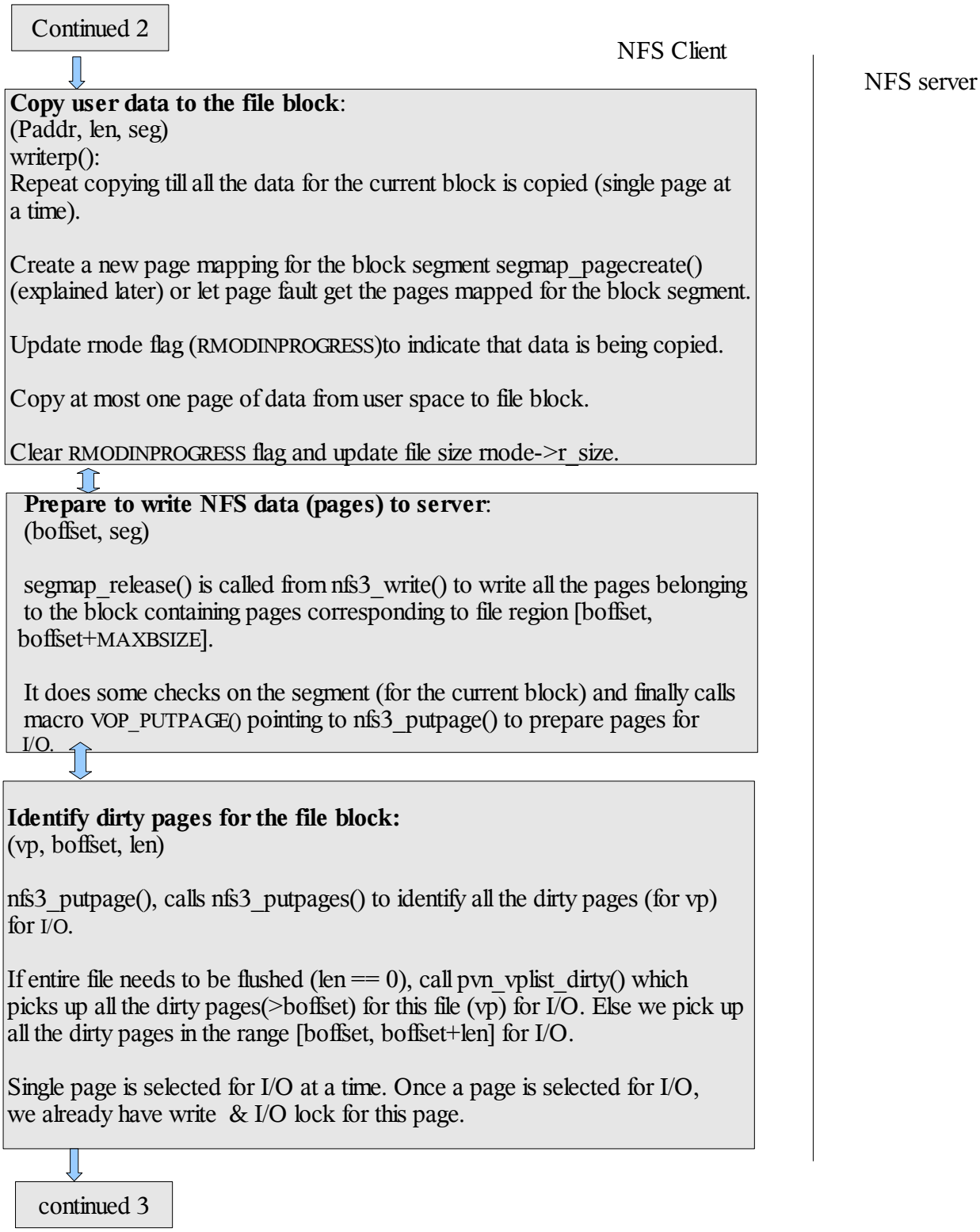
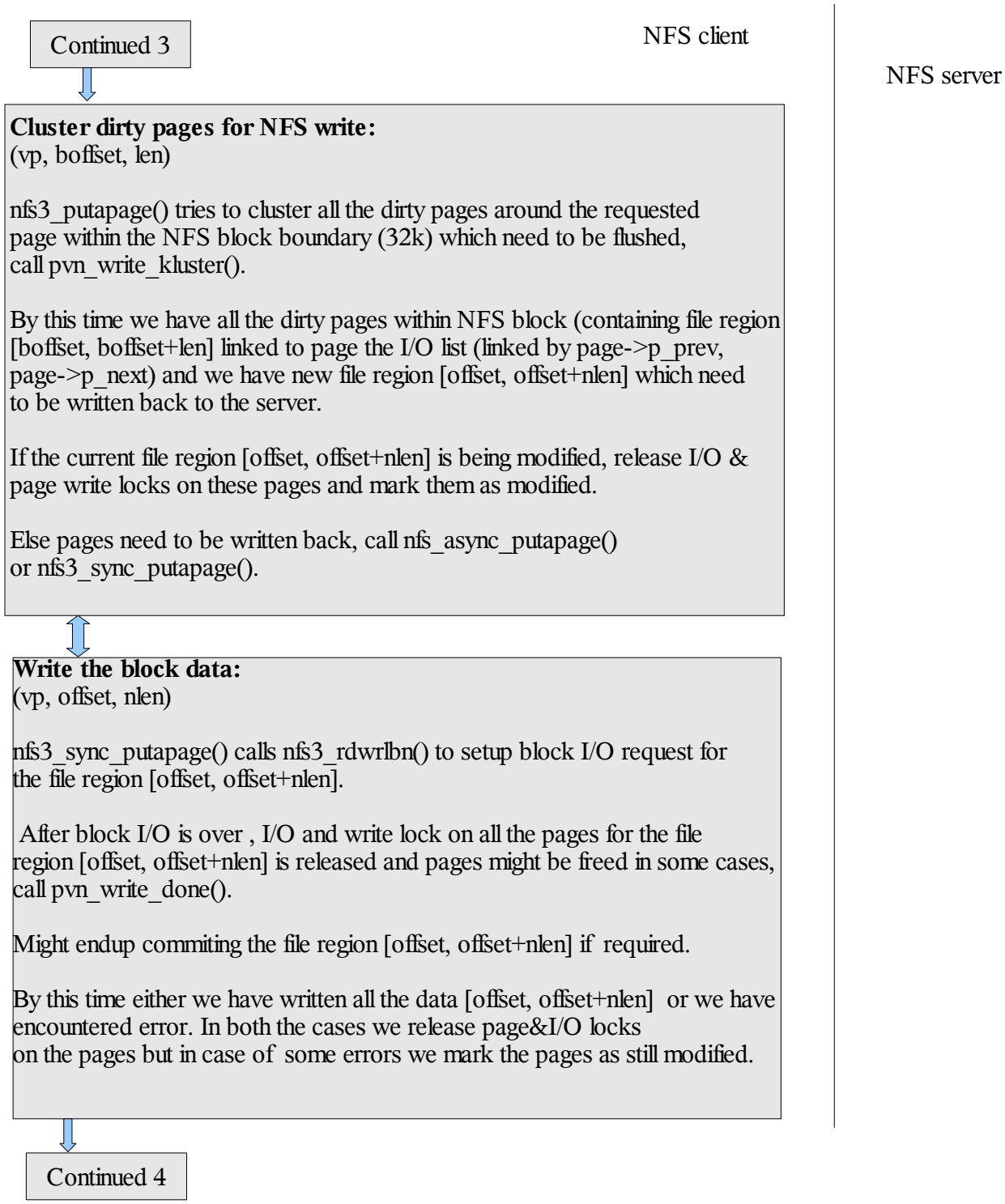


Fig. 7

2. In case caching is enabled or we are using client-side direct I/O with file mmapped with either cached pages, we need to go through kernel VM framework to allocate pages to contain the file data. Each kernel page for this vnode contains data for the combination of [vp, file_offset]. The pages are mapped to kernel virtual addresses allocated by the segment driver for each block of the file (for the combination of [vp, block_off]). Before writing the data, we have to read the page containing the current file offset requested by the user application and map at kernel virtual address specified by segment address for the block containing this page. We now copy the user data to the page (containing the file offset). Finally, we need to cluster all the dirty pages around this page (within NFS block boundary) and place NFS block I/O request to write requested portion of the file data to the NFS server using RPC call (Fig. 7).




Continued 2
NFS Client
NFS server
Copy user data to the file block:
(Paddr, len, seg)
writerp():
Repeat copying till all the data for the current block is copied (single page at a time).
Create a new page mapping for the block segment segmap_pagecreate() (explained later) or let page fault get the pages mapped for the block segment.
Update rnode flag (RMODINPROGRESS)to indicate that data is being copied.
Copy at most one page of data from user space to file block.
Clear RMODINPROGRESS flag and update file size rnode->r_size.
Prepare to write NFS data (pages) to server:
(boffset, seg)
segmap_release() is called from nfs3_write() to write all the pages belonging to the block containing pages corresponding to file region [boffset, boffset+MAXBSIZE].
It does some checks on the segment (for the current block) and finally calls macro VOP_PUTPAGE() pointing to nfs3_putpage() to prepare pages for I/O.
Identify dirty pages for the file block:
(vp, boffset, len)
nfs3_putpage(), calls nfs3_putpages() to identify all the dirty pages (for vp) for I/O.
If entire file needs to be flushed (len == 0), call pvn_vplist_dirty() which picks up all the dirty pages(>boffset) for this file (vp) for I/O. Else we pick up all the dirty pages in the range [boffset, boffset+len] for I/O.
Single page is selected for I/O at a time. Once a page is selected for I/O, we already have write & I/O lock for this page.
continued 3


Continued 3
NFS client
NFS server
Cluster dirty pages for NFS write:
(vp, boffset, len)
nfs3_putapage() tries to cluster all the dirty pages around the requested page within the NFS block boundary (32k) which need to be flushed, call pvn_write_kluster().
By this time we have all the dirty pages within NFS block (containing file region [boffset, boffset+len] linked to page the I/O list (linked by page->p_prev, page->p_next) and we have new file region [offset, offset+nlen] which need to be written back to the server.
If the current file region [offset, offset+nlen] is being modified, release I/O & page write locks on these pages and mark them as modified.
Else pages need to be written back, call nfs_async_putapage() or nfs3_sync_putapage().
Write the block data:
(vp, offset, nlen)
nfs3_sync_putapage() calls nfs3_rdwrlbn() to setup block I/O request for the file region [offset, offset+nlen].
After block I/O is over , I/O and write lock on all the pages for the file region [offset, offset+nlen] is released and pages might be freed in some cases, call pvn_write_done().
Might endup commiting the file region [offset, offset+nlen] if required.
By this time either we have written all the data [offset, offset+nlen] or we have encountered error. In both the cases we release page&I/O locks on the pages but in case of some errors we mark the pages as still modified.
Continued 4

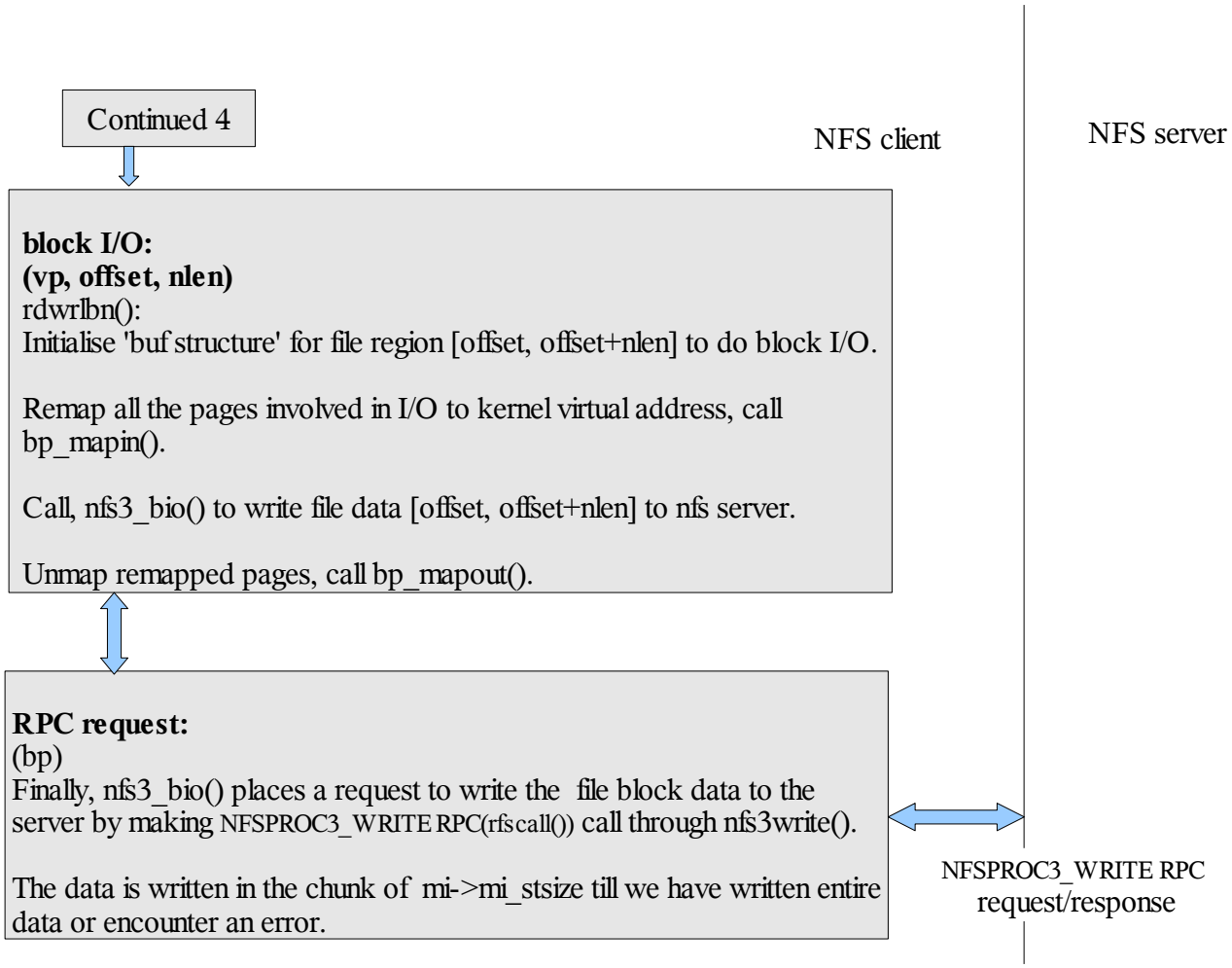

Continued 4
NFS client
NFS server
block I/O:
(vp, offset, nlen)
rdwrlbn():
Initialise 'buf structure' for file region [offset, offset+nlen] to do block I/O.
Remap all the pages involved in I/O to kernel virtual address, call bp_mapin().
Call, nfs3_bio() to write file data [offset, offset+nlen] to nfs server.
Unmap remapped pages, call bp_mapout().
RPC request:
(bp)
Finally, nfs3_bio() places a request to write the file block data to the server by making NFSPROC3_WRITE RPC(rfscall()) call through nfs3write().
The data is written in the chunk of mi->mi_stsize till we have written entire data or encounter an error.
NFSPROC3_WRITE RPC
request/response

Fig. 8

Mechanism of NFS read with caching enabled

Fig. 9 mentions sequence of important function calls to write NFS data:

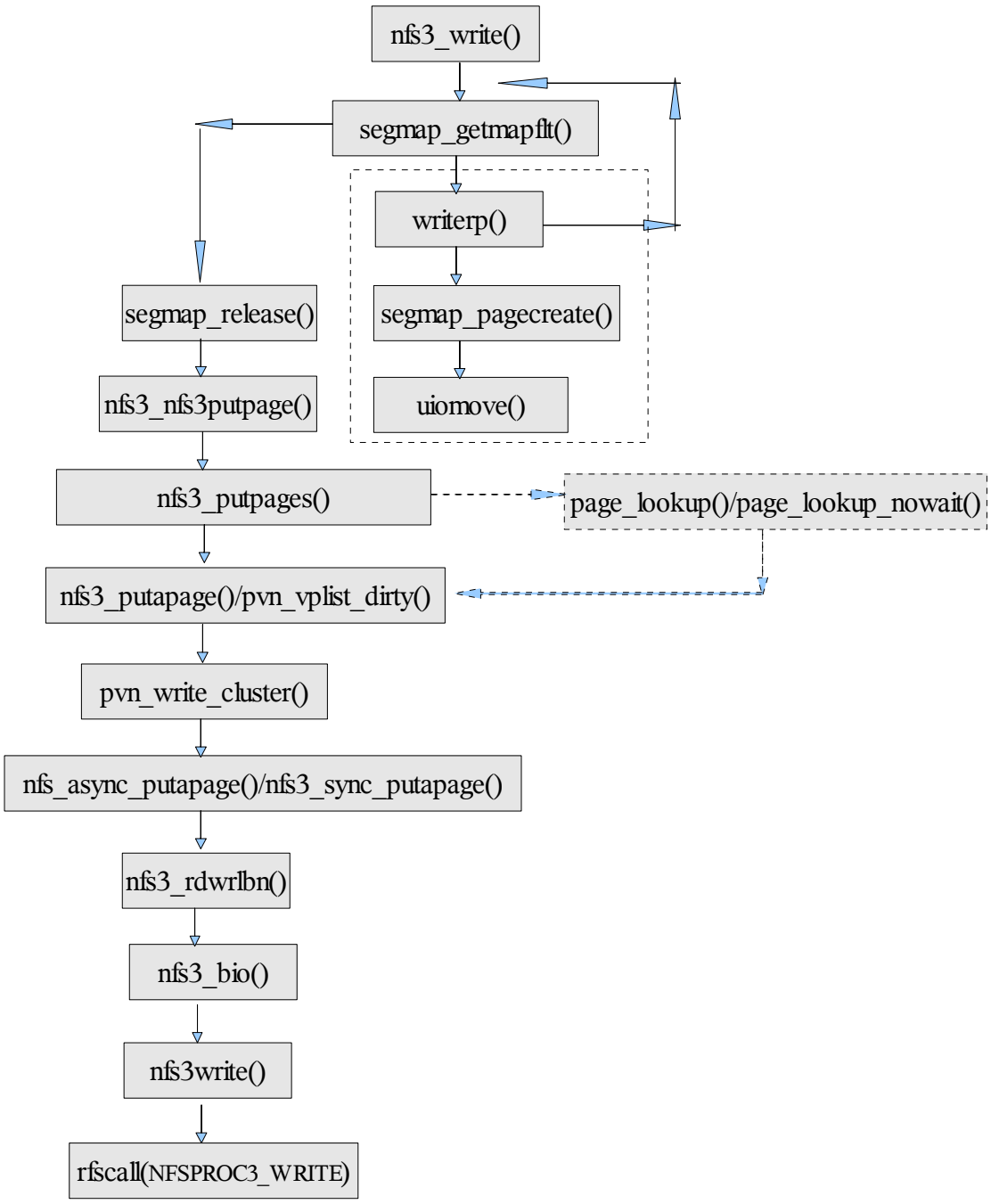


Fig. 9

Page & Block I/O in NFSv3 write operation:

We discuss here how NFS write request is processed by the kernel. User application requests kernel to write data to some portion of NFS file. Kernel copies this data to pages belonging to requested region of the file. The pages are marked as dirty. Single page is selected for I/O at a time. NFS data is written back to the server in block size of 32k. So, all the dirty pages around the page (selected for I/O) and within the current NFS block are then clustered together and then block I/O is done. All the data within the NFS block is written back to the server in small chunks by generating NFSPROC3_WRITE RPC call. Lets look at various steps involved in brief. The entire explanation we assume that we are processing ASYNC write request to write 1 block bytes(len=2*PAGESIZE) data from file offset 17000(f_offset) otherwise we mention the len and f_offset to explain certain situations.

User application generates write request for file(vp), to write 'len' bytes of data from offset 'f_offset'. We need to copy data from user space to kernel space. The data is copied in maximum chunk of block (MAXBSIZE). Let block offset for the block(MAXBSIZE) containing f_offset be 'boffset'. segmap_fault() gets kernel virtual address for the file block mapped to the segment [vp, boffset].

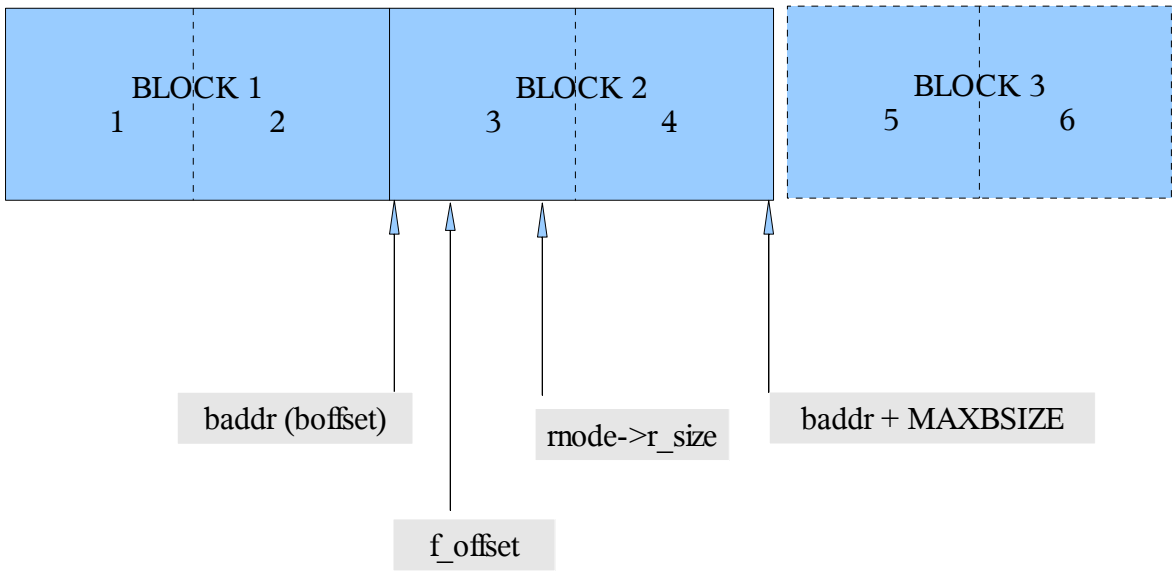


Fig. 10

PAGESIZE=8k
MAXBSIZE=2*PAGESIZE

File-block segment [vp, boffset] is mapped to the kernel virtual address range [baddr, baddr+MAXBSIZE]. From fig. 10 we can see that two pages(3 & 4) need to be mapped to segment [vp, boffset]. Each file(vp) page corresponds to 'PAGESIZE' size of file data at offset 'p_offset',

[vp, p_offset]. We need to get pages 3 & 4 and map them to the address range [baddr, baddr+MAXBSIZE]. writerp() copies one page of data at a time to the pages mapped for block [vp, boffset]. Before copying the data to the page, we need to make decision whether we need to get page data from the NFS server or not. We don't get page data from the server in the following conditions -

1. f_offset is at PAGESIZE boundary.
2. and PAGESIZE data needs to be copied or f_offset is beyond current EOF.

From Fig. 10 we can say that page needs to be created for the following combination-

1. f_offset=16384 and len= PAGESIZE or
2. f_offset=16384 and rp->r_size(clients view of page size) < 16384.

page 3 for block 2 is picked from the cache or created and mapped to baddr segment address in this case. This is done by calling segmap_pagecreate(). So, here we avoid reading in data for page 3 from NFS server. We will either get this page from the local cache or if there is a cache miss, we create one and write data to it.

In all other cases, page with valid data is either read in from the local cache or from the NFS server (in case of cache miss) and mapped to the segment address. For e.g., f_offset=17000 & rp->r_size=17500, we get page 3 with valid data either from cache or from NFS sever which is finally mapped to segment address baddr(because 616 bytes of data from offset 16384 is still unmodified and original data needs to be kept intact). This happens as a result of page fault (when baddr is accessed to copy data as the page is not yet mapped) where segmap_fault() is called. This calls nfs3_getpage() to get page 3 with valid data and will finally map the page to the segment address baddr.

By this time we have the page mapped at address baddr, the page is locked (shared) without I/O lock held on the page. The pages is linked in the page hash list and also in the vnodes vp->v_pages list.

We need to copy data from user application to the page. Before copying data, we mark the rnode flags to indicate that the file is being modified (rp->r_flags should be set to RMODINPROGRESS). At the same time we also store information about the file block which is being modified. We copy data to the page [vp, boffset] call uiomove(). Now we modify the file size (rp->r_size) to reflect the new file size(if written beyond the EOF) and RMODINPROGRESS flag is cleared from rnode's flag. RMODINPROGRESS flag is required to avoid any loss of data just written. There is a small window between the uiomove() & rp->r_size modification. In this window, any async thread (doing putapage operation) may interfere and find that the page [vp, boffset] is dirty. It picks this page for I/O and put it on the dirty list (dirty bit for the page is cleared when page is on the dirty list) but the file size is not yet modified. So, while doing final I/O, this page might be skipped as the I/O is done only for rp->r_size bytes of data. Finally after the I/O, the page is removed from the dirty list (even though the data from this page is not written back to the server) with dirty bit not set. So, this page is not be considered for I/O from segmap_release() and the data for the portion of the file corresponding to this page is lost.

We continue to copy data to the subsequent pages till either one complete block or requested len (whichever is minimum) of data is copied. For e.g., if the user has requested to write 16k bytes(len) of data from offset 17000 (f_offset), we finally have data in pages 3 & 4 which are mapped to address range [baddr, baddr+MAXBSIZE] (block 2 [vp, boffset]) when we return from writerp() which processes 1 block of data at a time(block 2). Since this is ASYNC request and one block of data is copied, we write out file block 2 (page 3 & 4). After processing page 3 & 4, block 3 data will be processed(for block 3, we need to write 616 bytes of data since 15768 bytes of data is required to fill block 2).

We check if the copied data needs to be written back to the server. If it is sync write operation, we need to write the copied data immediately, else we wait till we have at least written one block of data. To write data to NFS server we call segmap_release().

We repeat the entire cycle explained above till we have written all the data in a chunk of block(MAXBSIZE) size. For e.g., if the MAXBSIZE len of data is to be written from offset 17000, we need to get mapping for block 3, get pages 5 & 6 mapped for segment corresponding to block 3, copy user data to page 5 & 6 and finally write back data in these pages to the NFS server.

segmap_release() checks if the segment needs to be unmapped (not f relevance to the topic). It calls macro VOP_PUTPAGE() to write file data in the range [boffset, boffset+MAXBSIZE]. This macro points to nfs3_putpage(). nfs3_putpage() increments rnode's reference count rp->r_count and calls nfs_putpages() to further process the write request.

nfs3_putpages() selects the dirty pages for the vp in the range [boffset, boffset+MAXBSIZE] or entire vp pages(>boddset in the list (vp->v-pages). In case we are searching the entire vp->v_pages list for dirty pages(>=boffset), we call pvn_vplist_dirty() else we call nfs3_putapage(). Finally pvn_vplist_dirty() also calls nfs3_putapage().

pvn_vplist_dirty() checks for dirty page(>boffset) in vp->v_pages list, if found it picks it for the I/O, get's I/O lock on the page and calls nfs3_putapage() to further process the write request for this page. We arrange the pages in the vp->v_pages list depending on the vnode type. VMODSORT type vnode will have sorted list of pages arranged in vp->v_pages else pages are arranged in any order and a new page is always added to the head of the list. This means all the modified pages are added to the tail of the list in case of VMODSORT type vnode.

vp->v_pages is a doubly linked circular link list, we need to keep track of already visited pages in the list since pages can be added at the head/tail of the list is being processed. We insert two markers at the end of the list to keep track of the visited pages.

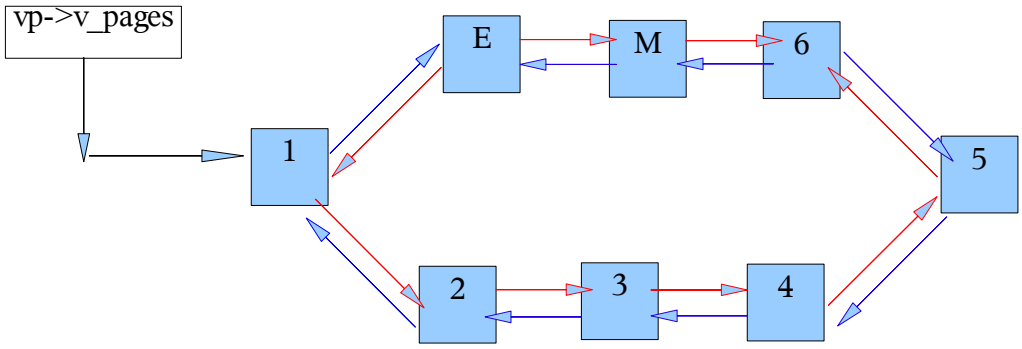


Fig. 11

page->p_vpnext

page->p_vpprev

E is the end marker.
M is the last one but marker.

As shown in Fig. 11 we have two marker pages at the end of the list E (end) and M (mark). List is traversed from tail i.e., page 6 in Fig. 11. If page 6 is the page to be processed (>=boffset & dirty), M and 6 pages swap their position in the list and page 6 will be processed. So, final positions will be as shown in Fig. 12.

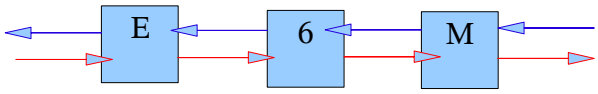
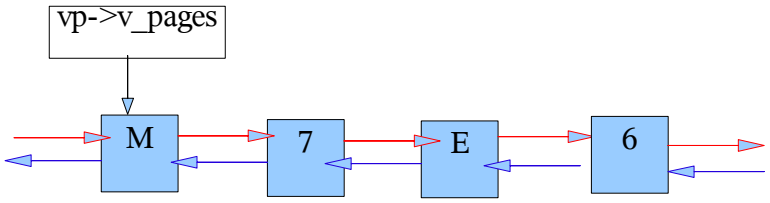


Fig. 12

This way we visit all the pages in the list and M moves one link ahead towards the head in each iteration. This loop breaks in following cases -

1. if it is VMODSORT type vnode and we find first unmodified page.
2. It is an ASYNC operation and M has reached head of the list (vp->v_pages == M) Fig. 13(a)
3. it is SYNC operation and M has reached end marker (M->p_vpprev==E) Fig. 13(b).

For ASYNC(a):



For SYNC(b):

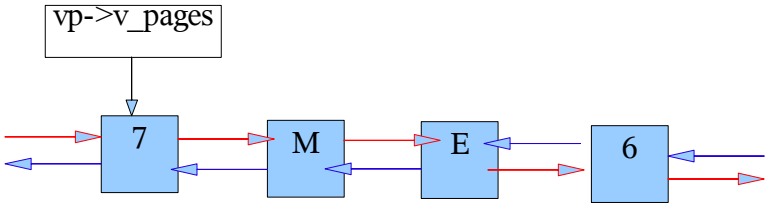


Fig. 13

We have selected the page for I/O [vp, boffset] and we have acquired SHARED & I/O lock on the page. We call nfs3_putapage() to further process the page for I/O.

nfs3_putapage() operates on NFS block size (32k). It finds the NFS block which contains file offset boffset. If f_offset is 17000 then we have 1st NFS block (with offset ==0) for current f_offset. So, it tries to cluster all the pages around page 3 (containing f_offset) tries to cluster all the dirty pages around page 3 [vp, boffset] (scan page 1, 2 & 4) within the NFS block boundary [0, 32k] (please refer Fig. 10). we call pvn_write_cluster() to do the clustering. pvn_write_cluster(), returns with the list of dirty pages (with page & I/o locks acquired) linked in the I/O list (page->p_prev & page->p_next) and a new io_offset and io_len for the file on which I/O needs to be done. Lets say we have page 3 & 4 clustered in our example so, the new file offset as 16384 and file length as 16384 on which I/O needs to be done.

We check if we have modified the file in the range range [io_offset, io_offset+io_len]. rp->r_modaddr keeps the file block offset for the block which is being modified in writerp(). If rnode flag (rp->r_flags) is set to RMODINPROGRESS, and rp->r_modaddr is within the file range [io_offset, io_offset+io_len], we will unlock all the pages in the I/O list, mark them as modified, mark file rnode as dirty and return. Else we call nfs_async_putapage() for ASYNC & nfs3_sync_putapage() for SYNC I/O for file [io_offset, io_offset+io_len] portion of the file (16384 bytes from offset 16384 in our example).

We discuss only sync operation as async I/O artchitecture is already explained in async NFS architecture topic. nfs3_sync_putapage() calls nfs3_rdwrlbn() to initialise block I/O request structure and place block request for [io_offset, io_offset+io_len] portion of the file. If nfs3_rdwrlbn() returns error(ENOSPC , EDQUOT, EFBIG, EACCES), and it is synchronous operation, we invalidate all the pages for file region [io_offset, io_offset+io_len] (page 3 & 4 in our case). Else we just release locks(I/O & write) on the pages and if required, generate the commit request to commit file data just written.

nfs3_rdwrlbn() initialises “buf” structure for block I/O in the range [io_offset, io_offset+io_len] and remaps this region of the file to specified kernel virtual address. It places block I/O request for writing data to the NFS server. nfs3_bio() is called to execute block I/O request. It calls nfs3write() to generate NFSPROG3_WRITE RPC (by calling rfscall()) to write data to the server in chunk of mi->mi_stsiz. On return from nfs3_bio(), we unmap the remapped portion of the file in the range [io_offset, io_offset+io_len]. This way we have written NFS data to the server in the file range [io_offset, io_offset+io_len] corresponding to page 3 & 4.

Cache Management:

This section explains the data & attribute NFSv3 client's cache management. Clients maintains it's own copy of the data and attribute caches for the file at the server. Client gets file attributes in two ways-

1. client requests the server for file attributes whenever it finds that the file attributes in its cache have expired.
2. Client gets pre & post modification file attributes in response to read/write/commit request from the server.

Client needs the file attributes in various places before read/write/open/close/create etc., calls nfs3getattr(). It first looks into the cache for attributes, calls nfs_getattr_cache(). If the cache attributes have expired, it calls nfs3_getattr_otw() to get file attributes from the server.

How does a client decide that he cache attributes have expired??
rnode->r_attrtime is the filed that keeps the time when the attribute for this is suppose to be expired. Whenever cleint gets file attributes from the server,it stores the attribute in the cache and then modifies the attribute expiration time in rnode->r_attrtime. This value is current time + delta. 'delta' depends on how frequently the attribute are gotten from the server.
If the client finds that the attributes are still valid, it returns the attributes from the cache. Attribute cache is maintained in rnode->r_attr (of type struct vattr). If the cached attributes have expired, fresh attributes are gotten from the server and then they are cached in.

Whenever client gets fresh attributes from the server, it compares the fresh attributes with the cached attributes. If they are found changed (which means he file is being modified at the server), the cached data for the file is invalidated so that fresh data can be gotten back from the server on next read/write.
nfs_cache_check(), does all the validation on the cached attributes. If RWRITEATTR is set for this files rnode, we ignore the attributes from the server(because the changes in attributes are because of our own write operation). Else it calls macro CACHE_VALID(), which compares modification time and file size of the file from the server and the cached in attributes. If they are found different then it checks whether this change in attributes is because of it's own modifications (write/truncate operations originated from client). If the rnode->r_flags is set to RMODINPROGRESS, it means that uiomove() is in progress so client's rnode->r_size is not consistent. So, client marks an indication so that next time when attributes are asked by the client, it gets the fresh attributes from the server, rnode->r_flags is set to RPURGECACHE. If the attribute changes are not because of our modifications to the file, client invalidates the data cache by calling nfs_purge_caches() (this invalidates all the pages in vp->v_pages for this rnode).

The fresh attributes are now cached in by calling nfs_attrcache_va(). This stores fresh attributes in the cache (rnode->r_attr) and also modifies the next expiration time for the cached attributes. It also, mark an indication that now we can trust the attributes from the server (reset RWRITEATTR in ronde->r_flags if it is set). RWRITEATTR flag indicates that we should not trust the server's attributes

because we have got the modified file attributes from the server as a result of our own modifications. if we don't have this flag, we would end up invalidating the caches after each write/truncate because of which would impact performance drastically.

Open-to-close consistency:

Client requests file attributes from the sever at file open/close time. It does because the file might be changed at the server between close and open. This is required because, if client has performed write operation just before close, the RWRITEATTR gets set (rnode->r_flags) and fresh attributes from the server's write response is stored in the cache(done by nfs3_checkwcc_data()). When client receives first response for getattr request in close, it ignores the attributes and resets RWRITEATTR. Next time when attributes are requested from the server (at the time of file open), we cache in the attributes gotten from the server and these attributes may have changed because somebody else has modified the file at the serve between open and close. If we don't request for attributes in close, we would still have old attributes and RWRITEATTR flag set for this rnode, and finally we would have ignored the file attributes from the server even if the file was modified at the server and file attributes have changed. In this case, client will never know that the file has modified at the server and it continue to use the old attributes for further operations causing inconsistency and file corruption.

Asynchronous I/O architecture:

Asynchronous NFS will allow many parallel read/write operations for different regions of the same file. This will enable clustering of write request from the queue before processing any of the other async I/O types. This will improve the performance and throughput of the overall write operation.

There are asynchronous NFS request queues for each NFS I/O types per mounted file system which are responsible for asynchronous I/O. nfs_async_start() is a function run as NFS async thread which will look for any async request queued for a specific mounted file system. This thread serves the queued request for each I/O type in a round-robin fashion. mntinfo_t structure consists of members to which will manage the async activities.

async request queue organisation:

Some of the members of this structures are:

struct nfs_async_reqs *mi_async_reqs[NFS_ASYNC_TYPES]:

array of async request queues for each I/O type. Where I/O types are -
NFS I/O types are:

NFS_READ_AHEAD,
NFS_PUTAPAGE,
NFS_PAGEIO,
NFS_READDIR,
NFS_COMMIT

struct nfs_async_reqs *mi_async_tail[NFS_ASYNC_TYPES]:
this is an array of pointers which points to the tail of the request queue for each I/O type.

struct nfs_async_reqs **mi_async_curr:
this points to the current async request queue which is being processed.

kcondvar_t mi_async_reqs_cv;

ushort_t mi_threads:
number of active async threads.

ushort_t mi_max_threads:
max number of async threads.

kmutex_t mi_async_lock:
lock to protect async list

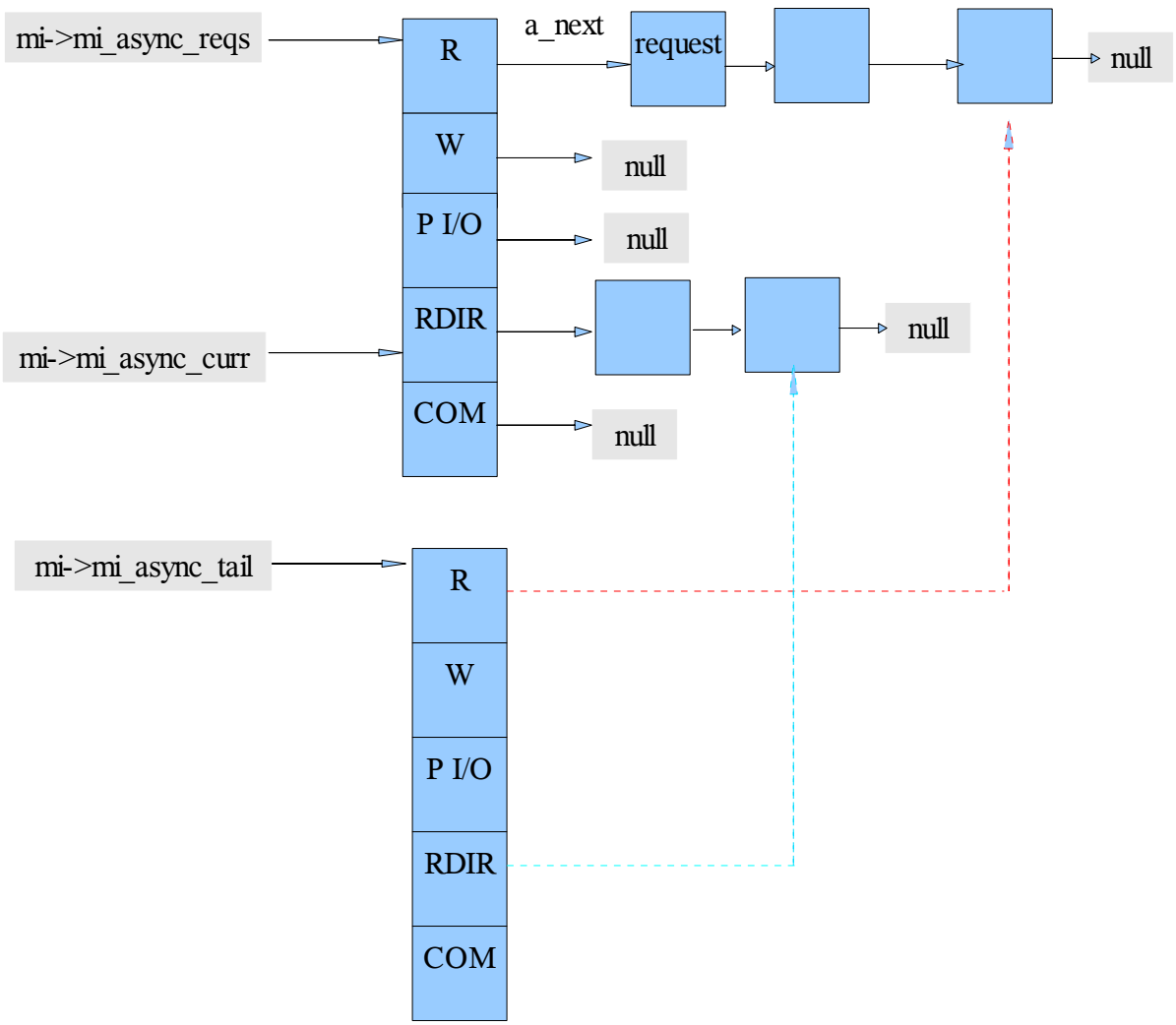


Fig. 14

(Note: Not shown pointers for all the mi_async_tail elements as the diagram will become more complicated. It is understood that W, P I/O & CO will be pointing to NULL because there are currently no requests queued for these I/O type in the request queue.).

R = NFS_READ_AHEAD,
W = NFS_PUTAPAGE,
P I/O = NFS_PAGEIO,
RDIR = NFS_READDIR,
COM = NFS_COMMIT

Processing of requests form the queue:

While processing the queue, mi->mi_async_curr is first accessed and checked if it is NULL. If not NULL and enough requests are not yet served from this queue, then the request is taken off the queue and processed and * mi->mi_async_curr will now have address of next node in the current list args->a_next. Else if it NULL, we move to the next queue in the request array with the following pointer increment:
++ mi->mi_async_curr;

By doing so, we are pointing to the next NFS async request Queue. We repeat the same till we have exhausted all the requests in the queue (Fig. 14). The queue list is synchronised with mi-> mi_async_lock lock.
Whenever a new async thread is created, we need to check if the mi-> mi_threads with mi->mi_max_threads. If former is lesser than the latter, we can create a thread else we cant because already maximum number of async threads are active. Else we create a new async thread nfs_async_start() and increment the mi-> mi_threads by 1. Each async NFS read & write request for the same file will correspond to different regions of the file which will never overlap. For e.g., file offset and file length for two different async write/read request will never overlap a file region. Whenever a new async request is generated for file read/write, rnode's members r_count & r_awcount are incremented by 1 to keep track of async activity on the file.

Data structures used for async operations:

nfs_async_reqs is a request structure used for each NFS async I/O. The main members of the structure are:

struct nfs_async_reqs *a_next:
pointer to next arg struct (NFS async request) in the request queue.

struct vnode *a_vp:
vnode pointer of the NFS file for which requuest is generated.

enum iotype a_io:
I/O type, listed above.

```
union {
        struct nfs_async_read_req a_read_args;
        struct nfs_pageio_req a_pageio_args;
        struct nfs_readdir_req a_readdir_args;
        struct nfs_commit_req a_commit_args;
} a_args;
```

These are argument structure for each I/O type. Async write operation are taken care of by nfs_pageio_req structure. Some important macros used for async NFS read/write operations (read/write args)-

For Readahead operation:

#define a_nfs_readahead a_args.a_read_args.readahead:
this gets the callcack function used to read block of NFS file from the server (nfs3_readahead() in our case).

#define a_nfs_blkoff a_args.a_read_args.blkoff:
this gets the block offset of the block to be read.

#define a_nfs_seg a_args.a_read_args.seg
This the segment to which this I/O belongs.

#define a_nfs_addr a_args.a_read_args.addr
This is the mapped kernel virtual address for the block where this block data should go.

For async write operation:

#define a_nfs_putapage a_args.a_pageio_args.pageio:
this contains the address of the callback function which will actually write data to the NFS server (nfs3_sync_putapage()).

#define a_nfs_pp a_args.a_pageio_args.pp:
this points to the list of pages involved in I/O.

#define a_nfs_off a_args.a_pageio_args.io_off:
This gets the file offset, which is the starting point for current write operation.

#define a_nfs_len a_args.a_pageio_args.io_len:
this gets the file length (starting at offset a_nfs_off) which needs to be written back to the server.

#define a_nfs_flags a_args.a_pageio_args.flags:
gets the flags for write operation (sync/async).

References:

The write-up is a result of reverse engineering and is prepared on the basis of research work done on the NFS V3 (client) source for Solaris 9.