

https://dspace.org/

Design of a disk-based file server involving the mapping of "pseudo-disks"

Hildyard, Mark William

2015-01-27
http://hdl.handle.net/10539/16783

Downloaded from a DSpace repository. https://dspace.org/

maintained, listing all track-groups available for replacement. A request involving a track in a group, results in the entire group being removed from the list. Only after the outstanding requests to tracks in the group have been satisfied, is the group placed at the rear of the queue. Groups with dirty sectors cannot be replaced, but are only added to the queue once they have been written to disk.

A disadvantage of this group-based replacement is in its utilization of cache space. The entire group remains in the cache for as long as any particular track is required, even though many of these tracks may not be needed. This inefficiency will be negligible, if the prefetch assumptions (Sect. 3.2.6.1) hold. An important side-effect is that generally, the larger the group the longer it remains in the cache (provided most of its tracks are ultimately used). This can be used to increase the likelihood of a cache hit for users simultaneously using shared disks.

3.2.6.3 *Write Strategy*

Using the cache, write operations can also be performed at RAM speed. To do this, write requests coming from clients are decoupled from the corresponding disk writes - the cache is updated with the new data, but only later is this committed to disk. This decoupling allows multiple write operations to a particular pseudo-disk to be accumulated, and for disk writing to be performed in the background.

Incoming data is allowed to *accumulate* in the cache. The dirty sectors are not written to disk, while it remains likely that there will be further writes to that area of the disk. When these sectors are finally written to disk, there will be a substantial reduction in the number of disk accesses compared with if they had been performed as individual writes.

Even if a write request arrives for a sector which is not already in the cache, it is not sent to disk. Instead, a group containing this sector is fetched to cache, anticipating that further writes to sequential sectors will take place.

Due to the decoupling of disk write operation from write requests, the point at which data is written to disk is non-critical with regards to the service as seen by the clients. Disk writing can therefore be performed as a *background* task. This allows uses of the disk (such as fetching data to cache), to take precedence - writing only being performed when the disk is free. Although demand for the disk for writing purposes is less immediate, it must be performed regularly to release cache space, since groups with dirty sectors cannot be replaced.

Again, as for prefetching and replacement, the cache write-strategy is group based. Each group must have a bit map recording the dirty sectors in each track of the group. A write queue lists all groups containing dirty sectors. Accumulation is achieved by allowing each group in this list a time delay from when it was last written to, before it can be flushed to disk. Writing is generally performed as a background task, but forced flushing takes place if there are too many groups in the write-list.

In summing up, the cache has a pseudo-disk track as its basic data unit. Essentially then, it is caching multiple pseudo-disks rather than the server's disk directly. The predominant characteristic is that of being group-based. From the perspective of the major strategies of prefetching, replacement and writing, a group or cluster of tracks is treated as a single entity.

Many of the decisions, particularly those of prefetching, rely on the assumption that the data requirements of users are largely sequential. This in turn requires contiguity both in the storage of files on the pseudo-disks, and in the storage of pseudo-disks on the server's disk. LRU replacement has been used, and the writing strategy involves accumulation and background flushing.

As far as the cache-size is concerned, the larger the cache the greater its advantages. As an indication of what constitutes a suitably large cache, consider the following:-

> There are 40 client machines, simultaneously using an average of 2 pseudo-disks each. Assume that for each pseudo-disk, there are simultaneously 2 groups of tracks in the cache, and that the average group size is 4 tracks. The track-size for the pseudo-disks is 4.5 Kbytes. This gives 40 x 2 x 2 x 4 x 4.5 = 2.88 MBytes.

Allowing for groups which are not being used, but which have not yet been flushed, a desirable size would be 3-4 MBytes.

3.2.7 File Server Management

The file server must allow additions, alterations and deletions of users and of common software. Part of this entails the alteration of the files with user and application information and the directory disk and requires off-line management. The actual transference of software to the server and its adaptation to the system, is best performed interactively from a client computer with the server in full operation.

Off-line management involves the editing of user and application records. It should allow new record entries and facilitate the input and modification of parameters. Pseudo-disks associated with new entries are not created during the editing process, but only when accessed by a client. Deletion of entries requires the deletion from disk of all pseudo-disks associated with the entry.

All modifications to the file of application records must be coupled with alterations to the directory file. The two files contain the same information, but whereas the record file is used by the server and oriented towards finding information given the application's number, the directory file is for use by the client to provide the application's number given the textual name. Information in the directory should be organized to facilitate this mapping and the hierarchical presentation of the available applications.

Since the directory file is produced at the file server and is dynamically modifiable, it is untenable to provide the file as a file on a directory disk - this would require construction of a

pseudo-disk for each operating system supported, from a detailed knowledge of the file system of that operating system (with reconstruction after a modification). Instead the file will be provided directly as the directory disk which is thus an improper pseudo-disk, and software at the client must obtain the information through absolute sector accesses.

After an application entry has been created during off-line management, and once the server is operational, the disks of the application can be copied from the floppy disk drive of a client computer (or from a user's private directory). This would be done by someone with manager status which has read-write access to all applications software. Some adaptation may be required, particularly for commercially obtained software. Firstly, if the application is classified as protected, it has to be executed immediately after attachment (*cf*: section 3.4.2). Thus a file with a standard name (for all applications) must be created on the first disk of the application, which when invoked executes the application. In addition, configuration may be necessary for packages which straddle more than one disk. When the application is mapped in, these disks are assigned to consecutive drives, whereas usually with a floppy disk system, floppy disks are physically swapped within the same drive.

3.3 REQUIREMENTS FOR THE EMULATOR

With the mapping of pseudo-disks to drives subject to dynamic alteration, it is imperative that the tracks held in the cache at the emulator for any particular drive are those of the disk currently attached to that drive (known at the file server). However it is undesirable that tracks should be deleted from the cache when a disk is removed, since if the disk is re-attached the tracks will have to be re-transmitted across the network. Also, tracks should not be held in the cache according to the drives to which they refer, since a disk could be detached and later re-attached to a different drive.

Instead, tracks held in the cache should contain a unique number, identifying the pseudo-disk to which they belong. A table containing the number of the disk currently attached to each drive, must be maintained and consistently reflect the file server's drive-map for this client. When a track is required the table must first be searched for the number of the disk attached to the required drive. The cache is then searched for the required track of this pseudo-disk.

For each alteration at the file server of a client's drive-map, a message must be sent to the emulator for it to perform the same alteration before any further I/O requests from the client are processed. This means that any request from the client which may alter the mapping of the disk drives must be atomic - i.e. nothing further can proceed until the status response to the request has been received.

Operations requested by the file server, will include switching pseudo-disks in and out of drives, and the deletion of the tracks of a particular pseudo-disk from the cache . Generally, disks are switched out rather than deleted. However, an explicit cache delete would be necessary if a disk were updated by another client, while it was not switched in. A user has a separate set of private disks for each operating system to which he has access, but these disks share the same identifiers (*cf* section 3.2.3). Thus, if a user changes operating systems, his user disks cannot merely be switched-out, but must be deleted from the cache.

3.4 SOFTWARE REQUIREMENTS AT THE CLIENT

Software which runs on the client machine, is needed to make the file service available to users. Requirements include the *client interface* for communicating with and invoking file server functions, and the *user interface* which presents and abstracts these functions in a user-oriented form. This software has to be written for each available operating system and provided on their system disks.

3.4.1 The Client Interface

The client interface has three levels - those of sector input and output, message communication and the invocation of server functions. The first level is transparent as it is already provided through operating system calls and through emulator requests to the file server. It requires no extra software at the client.

The second level provides the mechanisms for exchanging messages with the file server, as outlined in section 2.3.2. A routine must accept a message, copy this into a sector buffer, and perform an operating system call to write to the first sector of track 40. Another must perform an operating system read of the first sector of track 40, and extract and return a message.

The third level of the client interface presents an operation for each function available at the file server. This operation constructs the message structure required by the server from its parameters, sends the message and interprets the response.

3.4.2 The User Interface

Software comprising the user interface allows the user to choose parameters interactively and perform operations provided through the client interface. Current requirements are procedures for the loading of new pseudo-disks and for signing-on as a user.

The signing-on process prompts the user to enter his user number and password. It then transmits the request to the server and when it receives the response, displays a suitable message as to whether access has been granted. The 'Signon' program need only

be provided for the default operating system, and should execute immediately after the operating system has 'booted up'. Although no parameters are required, the 'sign-off' operation must also be made available in a user-executable form.

A Loading Program is required which allows new applications, user disks and operating systems to be loaded, or removed. The program should collect the parameters for the calls (Both a command and a menu driven interface should be presented), and provide an hierarchical presentation of the available applications and operating systems. To distinguish the applications accessible to the user, the loader should obtain the user's access category from the file server. However, access is determined at the file server and need not be enforced by the loader.

To present the user with a comprehensive directory service, the software must have knowledge of the structure and extract information from the directory or configuration disk - this is provided by the file server in the first drive following that of the system disk(s). The directory disk lists the available pseudo-disks, the operating systems and groups to which they belong and the type of access allowed for the user's configuration. It provides the correspondence between the textual names of pseudo-disks and operating systems presented to the user and the unique file numbers interpreted at the server. However, it is not strictly a pseudo-disk but rather a single data file at the file server. Thus it does not have a file structure, and must be read via direct sector accesses and not through operating system file operations.

When a user disk is required or disks are being released, the loader need only collect the parameters, send the request and report on the status. However, more is required when loading a new application or operating system. For loading a new operating system, if the request was successful, the program has to perform a 're-boot'. A new application may require immediate execution.

In loading a new application one of the parameters is the load-type, which is used as a protection mechanism. With the 'flat' load, all that is required is to send the request and report on its status. With the immediate or *protected load*, once a request has been successful, the application must be executed immediately. The code for this must remain co-resident with the application, and once the application has terminated and control has returned to the executing code, a request must be sent to the file server to remove the application.

4 DESIGN USING PROCESS-RESOURCE DECOMPOSITION

Two major non-concurrent jobs are required of the file server - interactive file serving, and off-line management. Management is concerned with the editing and updating of information on the users and the available applications software. Interactive serving is the primary purpose for the server, involving collection and processing of requests from nodes.

A data-flow design technique of *process-resource decomposition* [Walker, **16**], has been used for design and documentation. This technique is briefly introduced in the next section. The technique is then used to present the interactive serving process, with four successive levels of detail. At each level, the resources and processes are identified, and each is described at a very abstract level. The method is graphical, and process-resource diagrams for each level have also been collected together in Appendix A.

The resource descriptions define the behaviour of the resource and the operators which it presents to processes. Detailed data descriptions are not given (to prevent redundancy) - these can be found as part of the coding listings of Appendix B. Process descriptions define the behaviour and purpose of a process. A 'Pascal-like' programming descriptive language has been used to present process activities. A formal statement of 'syntax' for this language was not considered necessary.

4.1 THE PROCESS-RESOURCE METHOD

Rentsch predicts that "object-oriented programming will be in the 1980's, what structured programming was in the 1970's" [Booch, **17**]. Such an approach involves first isolating the objects in a system, where the concept of an object is a system component upon which certain actions can be performed. The object-oriented approach seeks to augment traditional functional decomposition methods (which focused on the abstraction of activities) to include the abstraction of data.

Booch identifies the limitations of functional development methods which object-oriented development attempts to address:

- Effective data abstraction and information hiding
- Exploitation of natural concurrency
- Adaptability to changes in the problem space

He also claims that object-oriented decomposition closely matches our model of reality, whereas functional decomposition is only achieved through a transformation of the problem space.

The *Process/Resource Approach* used in the design of the file server is one such object-oriented design method. Processes manage the data flow to and from resources, which manage data in particular data aggregates. The concepts of processes and

resources are equivalent to those of subsystems and monitors present in the DREAM design philosophy - see [Walker, **16**,pp3.1] for a discussion of similar methods.

A *Resource* consists of a data aggregate, together with a collection of operations which can be performed on this data aggregate. Resources are essentially passive in nature, responding only to external requests. A user's access to a resource is limited to these operations, and thus its internal structure of data is hidden.

Processes are the active entities of the system which manage the flow of data between resources using the operations provided by these resources.

The act of design entails the iterative *decomposition* of the system into processes and resources. At the highest level, a system consists of a single process with access to external resources, performing the entire systems activity. This single process can be decomposed into successive levels of multiple processes with local resources. The classification of a resource or process depends on the point of reference. For example from the point of highest level design the File Server is a single process accessing external resources such as the network, whereas from the point of view of the user the File Server is a resource which he can access.

With the decomposition of a system into a collection of independent activities, decisions on whether to serialize the processes or to use true or apparent concurrency must be made. The *mutual exclusivity* of access to shared resources is implicit. Once access has been granted, a requested operation must run to completion, before another process can access the resource. A criticism of extending the scope of mutual exclusivity to the whole resource, is that operations would tend to include instructions which make them well rounded from the point of view of abstraction, yet are not concerned with the shared data and thus need not be exclusive [Bishop, **18**,pp 6.10]. This is particularly relevant with true multi-processing systems. With apparent multi-processing (multi-tasking) systems, the wide scope of exclusivity is not undesirable, provided that processes cannot be suspended while accessing that resource.

The method is *graphical*. Detail in the symbols is limited to the name of the particular resource or process together with the operations being performed. Ellipses have been used to represent resources, rectangles for processes.

Refer to [**16**] for a detailed description of the Process-Resource method.

4.2 INTERACTIVE SERVING - 1st LEVEL DECOMPOSITION

At the very highest level, the file server, the network, and network software at the client's machine, collectively form a server resource, to which users have access. For design of the File Server however, we consider the highest level as just the file server element of this service - and its interaction with the network process at its own node.

At the highest level then, the file server can be viewed as a single process managing data flow between the external resources. These external resources are those comprising the network interface, and represent inputs from and outputs to the environment (*cf* Fig 4.2.1). The server receives request messages, and sector data for requests to update the sectors of a pseudo-disk. It must send messages responding to requests, and the track data of pseudo-disks in response to track-read requests.

4.2.1 1ST LEVEL: PROCESS BEHAVIOUR

The server dequeues requests, processes them and enqueues a response. If the request contained sector data (as in a request to write a sector) then the sector entry in the sector pool must be released. Enqueued responses must be monitored, so that post-processing can be performed on internal resources, once the responses have been sent by the network process. It has the following characteristic:

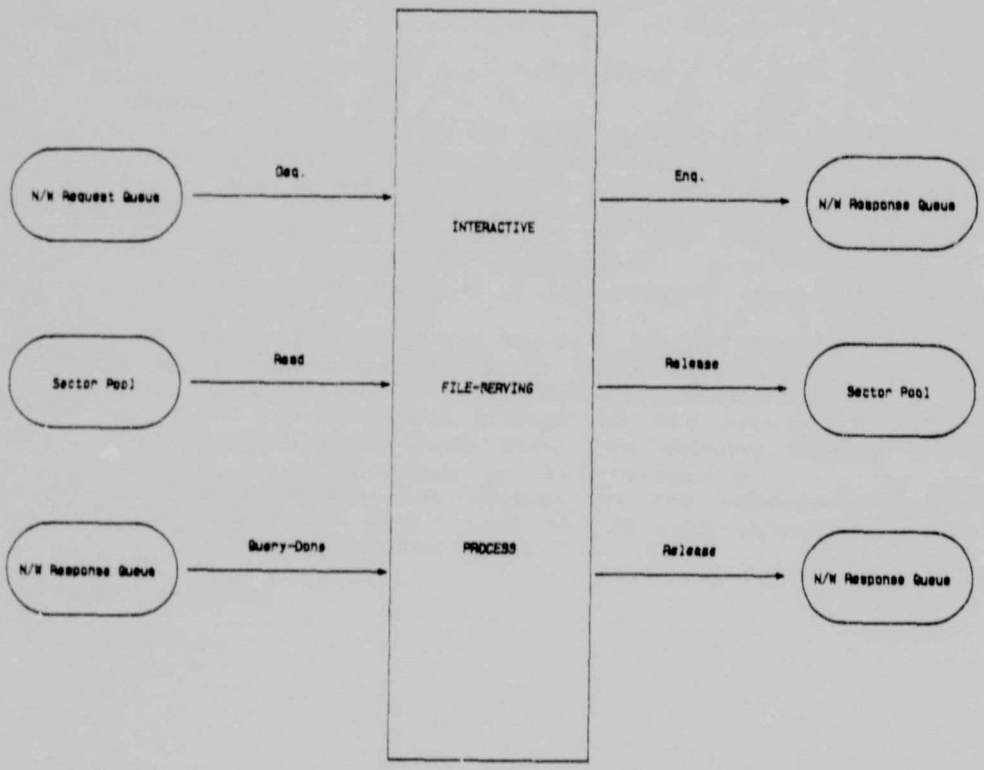
```
PROCESS: SERVER
  BEGIN:
    Repeat: Forever
      COBEGIN:
        * Dequeue a request from the Network input queue *
        CASE: * Request is to: *
           Send Track: * Get required track data *
                       * Construct an output message with
                           the location of this data *
                       * Enqueue the output message *

           Write Sect: * Copy data from the sector pool to
                           the required disk location *
                       * Release the sector pool entry *

           Other:      * Do internal processing on record for
                           a node e.g. change user or disks *
                       * Enqueue an output message *
        END: CASE

        * Look at the Network output queue *
        IF: * The network no longer requires the message *
           THEN: * Update internal resources *
                 * Release queue entry *
      END: Repeat
  END: Server
```

FIG 4.2 - INTERACTIVE SERVING: 1ST LEVEL



Abbreviations Used in Resource Diagrams

N/W - 'Network'
Enq. - 'Enqueue'
Deq. - 'Dequeue'
Various Ops - 'Multiple Operations - Identified at a lower level'
i/p Par: - 'Input Parameters'
o/p Par: - 'Output Parameters'
WL - 'Write-List'

Resources associated with the attachment of Pseudo-Disks:

These are the Common Information Resource, MD-Attachment Resource & the Opened-Disks Tables (both Application Map & Disk Map). The various operations on these resources are only identified at the lower levels.

For completeness, it is necessary to present the network process as to how it is intended to use the shared resources. The process must enqueue request messages and if the request has trailing sector data, it must obtain a sector entry from the pool, and add its location to the message. It must dequeue response messages and transmit them followed by track data if necessary, and afterwards mark the response as sent. It has the following characteristic:

```
PROCESS: Network
  BEGIN:
    Repeat: Forever
      COBEGIN:
        * Receive a request off the network *
        IF: * Request contains sector data *
            THEN: * Get an entry from sector pool *
                  * Copy data to the sector entry *
                  * Add the location of this entry to the
                      request message *
        * Enqueue the message on the Network Input Queue *

        * Copy entry from the Network output queue *
        IF: * The message includes track data *
            THEN: * Remove track information from copy *
                  * Send the message via the network *
                  * Read track data from server's memory *
                  * Send data via the network *
            ELSE: * Send the message via the network *
        * Dequeue the entry from the Network Output Queue -
            This marks the entry as sent *
    END: Repeat
  END: Network
```

4.2.2 1ST LEVEL: EXTERNAL RESOURCES

The network interface consists of four resources - two involved with message input/output (requests and responses), and essentially two with data input/output. *Messages* are sufficiently short to handle wholly within a queue format. However to avoid the inefficiency of large data relocations, the *data cannot* be queued as part of the message.

Arriving sectors cannot be handled in a queue. The server processes more than one node's request at a time, and these will not be completed in the order of their arrival. Since the sector data is required for the duration of the request (and no copy is made), the sector data blocks will be released in a different order to that of their arrival. Therefore, sectors received from the network should be placed in a pool of sector space until the server is ready to copy it into its cache.

Since disk data is to be cached, there is no specific data output resource. Instead, the network process is given direct access to positions in cache memory (specified in the output message). The network process does not have access to the cache resource as such (which is a complex, local resource of the server), - so it cannot release data in the cache once it has completed sending it. For this reason, the message output resource is a modified queue - the network doesn't consume an entry, but marks it as having been sent, allowing the server to re-look at the message and perform functions on the cache before the message is released back to the resource.

Since the resources are external, some operations are performed on them by the file server and others by the network process which links the server machine to the network. This can result in contention for the resources. No decision can be made at this level on access resolution, since it depends on whether the implementation is in a multi-processor, multi-tasking or single task environment.

4.2.2.1 Network Request Queue

The Network Request Queue is a standard FIFO queue containing request messages from all nodes. The network process receives messages and performs the enqueueing operation, and the file server dequeues requests.

Data Description

The data structures include:

- Request$Array: An array of fixed length message entries. The first byte of a message contains the source node, and the rest is the request.
- Read$Pos: A pointer to the front entry in the queue.
- Write$Pos: A pointer to the next entry which is available for storing a new message.
- Q$Size: The current number of messages in the queue.

Operators

Initialize: Initializes all variables as for an empty queue.

Enqueue: Copies a message into the next available message entry, increases the queue size and increments the write position. (The operation is used by the network process).

Dequeue: Copies a message from the front entry, decreases the queue size and increments the read position.

4.2.2.2 Sector Pool

The sector pool provides a temporary store for sector data arriving off the network, until the server can process a request and write the data to disk - i.e. it helps decouple the network process from the server. It consists of a pool of data blocks each the size of a sector in the pseudo-disks, together with an array recording the location of each data block, and linking free blocks into a spacelist.

Data Description

The data structures include:

Data Blocks: A number of fixed-size blocks of memory. (Each block should be contiguous, but it is not necessary for separate blocks to be contiguous with one another).

Sector$Pool$Array: An array with an entry for each sector data block. Each entry contains:

- A pointer to the data block.
- A pointer to another item in the array (used to implement the space list).

Space$List$Rock: Points to the first entry in the array, whose associated data block is free.

Operators

Initialize: This creates the data blocks, writes their locations in the array, and links all entries in the sector pool into the space list.

Get-Entry: If the space list is not empty, the first entry is removed and a pointer to that entry returned to the caller. (The operation is used by the network process).

Release: The specified sector entry is inserted at the front of the space list.

4.2.2.3 Network Response Queue

The Network Response Queue is a queue of fixed length response messages, awaiting transmission to client nodes. It is a modified FIFO queue in that the dequeueing operation doesn't consume an entry. Instead it marks the entry as sent and a separate operation releases the entry for use on the producer side. This allows the server to re-look at an output message and perform post-processing on its internal resources (specifically the cache). Therefore, the server process both produces and releases entries in this queue.

Data Description

The data structures include:

Response$Array: An array of fixed length message entries. The first byte contains the source node, and the second byte indicates whether the network has transmitted it yet. The rest is the response message.

Read$Pos: A pointer to the front entry in the queue.

Done$Pos: A pointer to the first entry in the queue with which the network process is already complete.

Write$Pos: A pointer to the next entry available to store a new message.

Q$Size: The current number of messages in the queue.

Qty$Free: The quantity of free entries not currently in the queue.

Operators

Initialize: Initializes all variables as for an empty queue.

Enqueue: Copies a message into the next available message entry, increases the queue size, decreases Qty$Free, and increments the write position.

Dequeue: Copies a message from front entry, marks the entry as 'done' (byte 2), decreases the queue size and increments the read position. (The operation is used by the network process).

Query$Done: Determines whether the output message at the done position has already been processed.

Release: Clears the entry at the current done position, increases Qty$Free, and increments the done position.

4.2.2.4 Data Output Resource

As mentioned, there is no specific data output resource, and track data must be obtained by the network, directly from the cache. The network process has direct access to the server's cache locations, but not to the cache resource as such. Essentially, the Data Output resource has as its data aggregate the file server's memory (without a definite structure imposed), while the only operators available, are those of reading and writing to memory. The location to be read is obtained from the message entry in the Response Queue.

4.3 INTERACTIVE SERVING - 2nd LEVEL DECOMPOSITION

Six processes can be isolated at the second level. The *Initialization Process* is responsible for initializing all the local resources and initiating the other processes. These other processes operate concurrently, requiring access resolution for all resources at this level. They are:

- A *Queue Producing process* which collects the input requests, and re-queues them in queues for each individual node.
- *Individual-Node Server Processes.* Each node has a server process which receives and responds to its requests, and keeps record of the current user and of the disks connected to drives.
- A *Disk-Read Process* which fetches pseudo-disk tracks from disk to the cache.
- A *Disk-Write Process* which writes all 'dirty' tracks in the cache to the pseudo-disk files.
- A *Complete-Sends Process* which updates the cache once a requested track has been successfully transmitted by the network.

These processes are illustrated in Figure 4.3 (a-f).

4.3.1 2ND LEVEL: PROCESS BEHAVIOUR DESCRIPTIONS

4.3.1.1 Initialization Process (Fig 4.3a)

This process initializes the local and external resources, loads configuration information and then initiates the concurrent processes (i.e. all except for the server processes). It has the following characteristic:

```
PROCESS:  Initialization
  BEGIN:
     * Initialize external resources *
     * Initialize 2nd level resources *
     * Read file with configuration information *
     * Configure system *
  END:  Initialization
```

4.3.1.2 Queue-Producing Process (Fig 4.3b)

The Queue Producing process is used to decouple the server tasks from the network input queue. It removes a request from the input queue and re-queues it for a specific node in the node-queues resource. This means that delays in processing one node's request, do not delay other nodes. If a request is received for a node, for which there is currently no serving task, a new serving task will be created. It has the following characteristic:

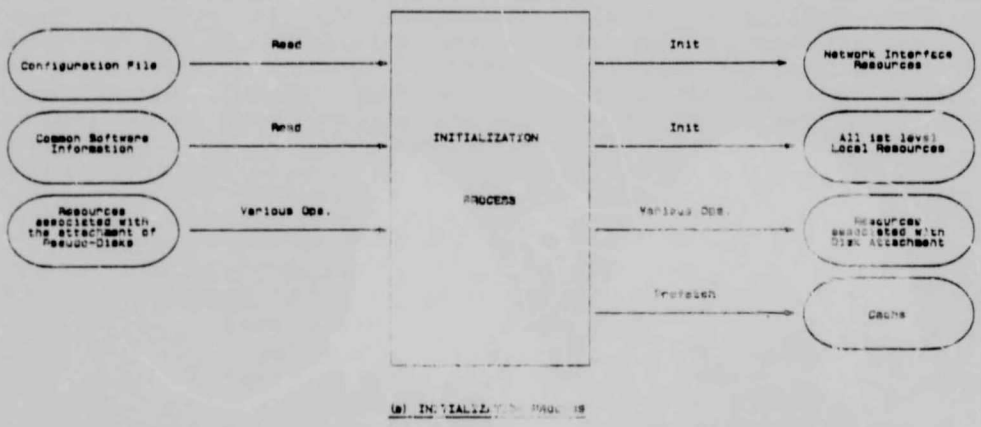
```
PROCESS:  Produce  Queues
  BEGIN:
     * Dequeue input message *
     * Determine which node it is from (first byte) *
     IF: * Node does not yet have its own serving task *
       THEN:
         * Create new Server task *
         * Send Node$Number *
         * Wait until new server task has received number *
     * Enqueue message in Node's Queue *
  END:  Produce  Queues
```

4.3.1.3 Individual-Node Server Process (Fig 4.3c)

There is an instance of this process for every active node on the network. Once initiated, it reads the number of the node which it will serve, and initializes internal resources. It then starts reading requests, processing them, and returning output. It has the following characteristic:

```
PROCESS:  Individual-Node  Server
  BEGIN:
     Read  (Node$Number)
     * Inform Producer process - Node number has been read *
     Repeat:  Forever
       BEGIN:
          Dequeue  (Node-Queue)
          * Interpret and process request *
          * Build and send message if required *
       END:
  END:  Individual-Node  Server
```

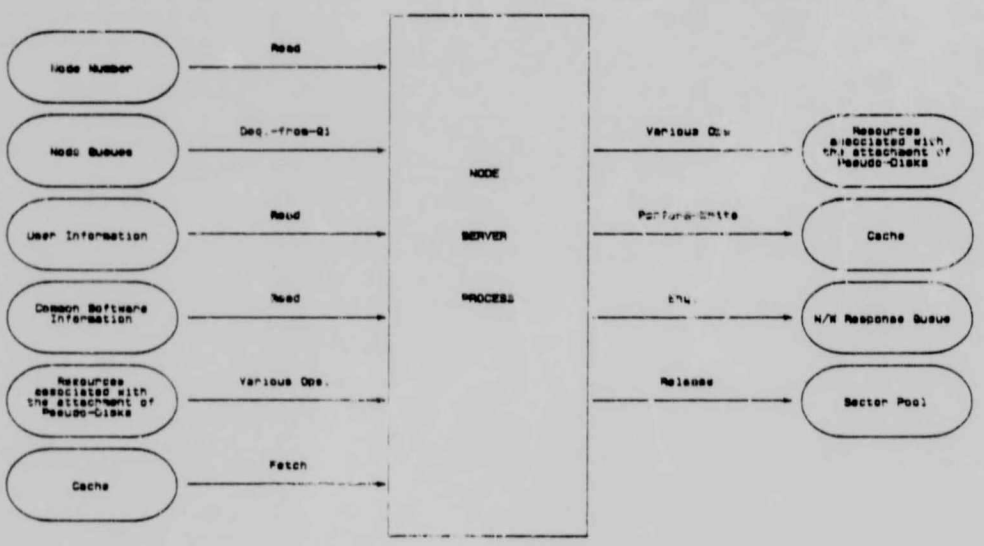
FIG 4.3 (a-c) - INTERACTIVE SERVING: 2ND LEVEL PROCESSES



(a) INITIALIZATION PROCESS



(b) [illegible]



(c) INDIVIDUAL-NODE SERVER PROCESS

4.3.1.4 Disk-Read Process (Fig 4.3d)

The Disk-Read process handles all disk reads of pseudo-disk file images, required by the cache. It receives a message from the cache through the Disk-Read Queue. The message will be to read all pseudo-disk tracks in a group determined by the cache (for optimizing disk usage). However, after a track has been read, it is immediately made available in the cache, so there is no lost efficiency in waiting for the whole group. It has the following characteristic:

```
PROCESS:  Disk-Read
  BEGIN:
     Repeat:  Forever
       BEGIN:
          Read  (Disk-Read  Queue)
          * Interpret  Message *
          Repeat: * for each track required *
            BEGIN:
               * Read the track from the Pseudo-Disk File
                    image to the location in the cache *
               * Mark Track present in the Cache *
            END:
       END:
   END:  Disk-Read
```

4.3.1.5 Disk-Write Process (Fig 4.3e)

The strategy here is to wait until the cache is ready with write information. The process then takes the front entry in the writelist, and determines to which disk it belongs. It then steps through each group entry in the cache for that disk, and writes all dirty sectors to the disk's image file. This should be efficient on the assumption that the pseudo-disk file images are nearly physically contiguous. It has the following characteristic:

```
PROCESS:  Disk-Write
  BEGIN:
     Repeat:  Forever
       BEGIN:
          * Wait until Cache is ready for writing *
          * When ready, use the group number obtained from
              Cache$WL$Ready to obtain the disk number which
              must be written *
          Repeat: * For each group entry of this disk *
            BEGIN:
               * Obtain from cache Next Group to be written *
               Repeat: * For each track in the group *
                 Repeat: * For each sector in the track *
                   IF: * Sector is dirty *
                     THEN: *Write sector to position in file*
            END:
       END:
   END:
```

4.3.1.6 Complete-Sends Process (Fig 4.3f)

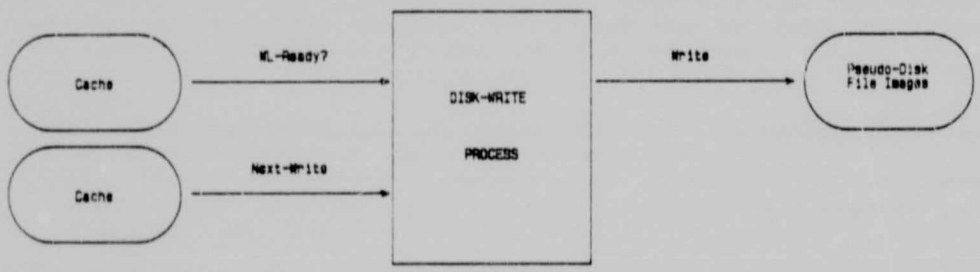
This process does post-processing on the File Server's internal resources, after the network process has already finished with the response data. Specifically this means that for all track read requests which have been sent, the process will reduce the quantity of outstanding operations for that track group in the cache. (No other requests require post-processing). Message entries must then be released back to the Network Response Queue. It has the following characteristic:

```
PROCESS:  Complete-Sends
  BEGIN:
     Repeat:  Forever
       BEGIN:
          * Wait until an entry in output queue is "done" *
          IF * Response is to a track read request *
             THEN: * Reduce outstanding operations at the
                     cache for this track group *
          * Release message entry in response queue *
       END:
  END:  Complete-Sends
```

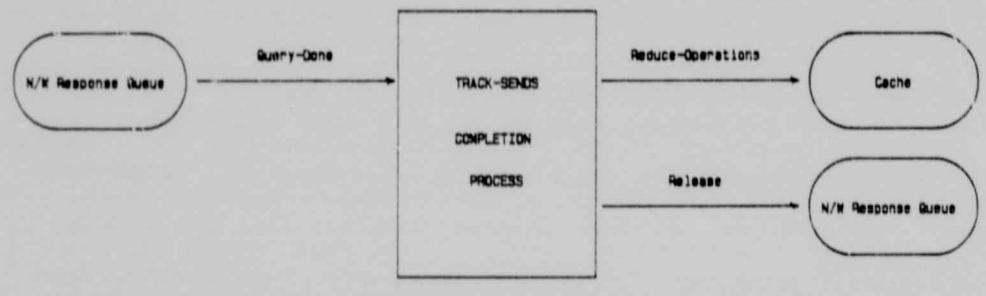
FIG 4.3 (d-f) - INTERACTIVE SERVING: 2ND LEVEL PROCESSES



(d) DISK-READ PROCESS



(e) DISK-WRITE PROCESS



(f) COMPLETE -SENDS PROCESS

4.3.2 INTERACTIVE SERVING: LOCAL RESOURCES

Resources identified at the second level are those internal or local to the first level serving process. They are therefore available as external resources to all processes identified at this and subsequent levels.

4.3.2.1 Configuration File

This file is read at initialization, and would contain any information used to provide flexibility and optimize performance, by changing run-time characteristics. Currently it would contain the file number for the default operating system and the file numbers of the most often used common disks. This saves processing time on well-used disks, since a user never has to wait for a disk/application to be opened or closed, and information is never flushed from the cache.

Data Description
The data structure consists of a file whose first entry is the file number for the Default Operating System. The second entry contains the quantity of files which are to be kept permanently open, and is followed by a list of file numbers.

Operators
- Read: This operation is the basic file-read operation and would be provided by the operating system.
- Edit: The file server requires an off-line editing process which must provide a facility to edit configuration data. This has not yet been considered.

4.3.2.2 Node Queues

The Node Queues Resource contains a queue for each possible node in the network. It decouples the individual server processes from the network input queue - and therefore from one another. The total quantity of entries (i.e. pool-size) is fixed, but there is no limit on the queue length for any particular node.

Data Description
The entries for these queues are drawn from a common pool, and a single space list links the unassigned entries. Data structures include:
- Node$Rqsts: A fixed-size pool (array) of request entries. Each entry contains a message and a pointer to another entry. The pointer is used to link entries into one of the node queues or the space list.
- Node$Queues: An array with an entry for each possible node. Each entry contains an identifier of the node's server process, and pointers to the first and last request entries currently in the node's queue.
- Space$List$Rock: A pointer to the first free entry in the pool.

Operators

Initialize: The queues for all nodes are initially empty with all entries linked into the space list.

Enqueue-to-Q_i: An entry is obtained from the space-list and inserted at the rear of the queue for the ith-node. The given data item is then copied into this entry.

Dequeue-from-Q_i: A data item is extracted from the entry at the front of the queue for the ith-node, and the entry is returned to the space-list.

4.3.2.3 HD-Attachment Interface

The Hard Disk Attachment Interface is a resource providing functions specifically for dealing with pseudo-disk image files. Generally, before the information in files can be accessed, a connection has to be created to the file and the connection opened. This resource is intended to supplement functions already provided in the file server's operating system, which relate to the attachment and opening of files. As such it is dependent on the architecture of the server's operating system, and what it includes therefore depends on the implementation. Its main purpose is to create a connection to the file and to create the file if it does not exist. This could entail creating directories within the path-name. When a file is created, it must be given the characteristics of a blank formatted disk in the operating system in which it is to be used.

Data Description

The resource performs operations on the Hard Disk(s) of the File Server machine. Specifically, it must implement the directory structure chosen for storing pseudo-disk files on the hard disk (*cf*:section 3.2.1). The resource also contains a series of New$Disk$Files - a file for each user operating system which is supported, containing the file image for a blank formatted disk.

Operators

Attach$Create$Dir: Creates a link to a directory or creates the directory if it does not exist.

Open$Create$File: Performs the necessary functions (e.g. opening) on a file for it to be used by the file server. It creates the file if it does not exist, and copies the file image for a blank formatted disk, to the new file.

4.3.2.4 Common Software Information

This resource is a database of the available operating systems and common pseudo-disks. Application entries provide the server with the mapping between the application file number (known to the client) and its position on the server's disk, together with access and other information. Operating system entries map an operating system number with an application entry identifying the software for supporting that operating system.

As discussed (*cf*:sections 3.2.7, 3.4.2), this database needs to also be presented to the user via software at the client - which maps application and operating system numbers to textual descriptions - to aid a client in requesting software.

Data Description

The database is a file with a list of operating system entries, followed by a list of application entries. Control variables define the positions of the first and last record in each list.

Control Variables:

Large$OS: The quantity of Operating System (O.S.) entries.

Large$Common: The highest numbered application.

First$Common$Entry: The offset in the file for the first user entry.

OS$Entry - The record number (from the offset in the file) is the Operating System number. An entry consists of:

Appl$Num: An operating system has an application associated with it, locating the pseudo-disks containing the operating system.

Name: The name of the operating system. It is used by the client when selecting an operating system. The name is also used in locating any pseudo-disk which runs under this operating system.

Application$Entry - The record number (from the [illegible] the file) is the Application file number. An entry [illegible]

OS$Num: The O.S. under which this applicati[illegible]

Group$Name: Applications are grouped accordi[illegible]ction, to aid the client when requesting disks.

Appl$Name: The name of the application - for use by the client when requesting disks. Together with the operating system name and the group name, it identifies the location of the application on the server's disk.

Assigned: If not assigned, then the application does not exist. This allows applications to be removed without re-packing the database and for software to be temporarily unavailable at certain times.

Qty: The quantity of disks in this application.

Max$Users: The maximum number of users for an application can be specified. This limit is applied on each disk in the application.

Access$Bytes: This contains two entries for each category of users, specifying whether users are allowed access to this application, and whether the access is for both reading and writing or read-only.

All$Reqd: This specifies whether a an application is dealt with as a unit (i.e. a user must have all disks connected at once), or whether isolated disks can be connected.

Load$Type: If 'immediate', then an application must be immediately executed at the client after attachment.

Track$Group$Size: This specifies the optimum size (the number of pseudo-disk tracks) to be fetched, when the cache fetches data for this application. A disk with large files on it would be given a large Track$Group$Size, while one with lots of very small files could be given a small Track$Group$Size.

Operators

Initialize: Initialization prepares the resource, reads the size information at the start of the file, the list of available operating systems and the first application entry.

Read: This tests to see that the requested application exists and that it is assigned and then reads the application's record from file.

Attach: A link to the application's location on the server's disk is created for the current record entry. The location on disk is constructed from the name of the operating system to which the application belongs, the group name and the application name (*cf*: section 3.2.1 and Fig 3.2.1.1).
Edit: A facility is required for adding entries, and for maintaining current information in the database.

4.3.2.5 User Information

This is a database of the users. It maps a user number to a record of the characteristics of that user. The user number is the record number of the user entry.

Data Description
The database is a file with a list of user entries. Initial variables define the positions of the first and last record.

Control Variables:
Large$User: The highest assigned user number.
First$User$Entry: The offset in the file of the first user entry. This allows control variables to be stored at the start of the file.

A *user record* contains the following items:
Password: This is used to authenticate a sign-on request.
Assigned: If not assigned, then the user does not exist. This allows users to be removed without re-packing the database or for users to be switched out of the system temporarily.
Access$Config: This defines the user's category which determines which application software is available.
Max$Qty: The quantity of user disks assigned to the user.
Init$OS: The operating system which must be loaded when the user first signs-on.

Operators
Initialize: This prepares the resource and reads the size information at the start of the file.
Read: This tests to see that the requested user number exists and that it is assigned, and then reads the user record from file.
Edit: A facility is required for adding entries, and for maintaining current information in the database.

4.3.2.6 Opened Disks Tables

This resource forms an intermediate link between the pseudo-disk at a user and the file-image at the server's disk. User drives are not connected directly to a file, which would involve multiple connections to the same file, but to an entry in this map. This entry has a single connection to a pseudo-disk file image - there is only one disk entry for each file in use.

The disk entry also provides the link to data in the cache, where data is not stored according to the file it derived from, but according to the open disk entry for which it was fetched.

All requests by a user for attachment/ detachment of pseudo-disks must access this map, which keeps account of which pseudo-disks are in use, and holds their access information. Since multiple connections to the same disk entry are possible, a new disk entry will be used only if there is no existing entry for that pseudo-disk file image. Each entry therefore keeps count of how many user connections it is serving. A request for detachment will not result in clearing of the disk entry, unless there is no remaining user connected to it.

The resource is complicated by two factors. There is the dualism of the 'application' and 'disks', due to the limitation of the pseudo-disk size not been enough to hold an entire application. Secondly, not all disks in an application need be used by a user. Therefore, in addition to the disk-map (above), there must be an application map, tracking the applications in use, and storing information relevant to the whole application or group of disks (such as access information and which of its disk are in use).

Data Description

The application and disk maps, map a file-number (which uniquely identifies a pseudo-disk), to an application/disk entry respectively. An application entry contains the characteristics of a group of disk entries, and the offset in the table is the Open-Application-Number. A disk entry is the link to the pseudo-disk location, and the offset in the table is the Open-Disk-Number.

The file-number is used when disks are requested, for locating the relevant disk entry, or for locating information from the Common$Info file and creating a disk entry. Afterwards it is the disk entry which is referenced by the system. Note that each node server has a drivelist, which links a drive to one of these open-disk-numbers. The cache also links its disk data to an open-disk-number.

Application Map Structure

A fixed-size array of entries storing data on each application currently in use. This information pertains to the application as a whole i.e. possibly a group of pseudo-disks.

Map control-variables include:

- First$Free$Appl: A pointer to the First un-used entry. The lowest entries in the map are assigned first.
- LargeOpAppl: Highest numbered entry in use. This makes searching more efficient, since only entries up to this point need be scanned.

Each entry contains the following:

- Appl$File$Num: The file number uniquely identifying the application to which the data pertains.
- Appl$Connection: A link to the application's position on the file server's disk.
- OS$Num, Qty, Max$Users, Access, All$Reqd, Load$Type and Track$Group$Size. These are a copy of the application's characteristics, as read from the Common$Info resource. A description of each item is given in the description of the *Common Software Information* resource (*cf*: Section 4.3.1.4).
- Qty$Open: The quantity of the application's disks currently in use.

Disk Map Structure
A fixed-length array of entries storing data on each pseudo-disk currently in use.
Map control-variables include:
First$Free$Disk: A pointer to the First un-used entry.
LargeOpDisk: The highest numbered entry in use.
Tot$Free$Disks: The quantity of un-used entries.
Each entry contains:
Disk: A file number identifying the Pseudo-disk in use.
Op$Appl$Num: The offset in the Application Map of an entry for the application to which the disk belongs (draws its access and other characteristics from).
Open$Count: A counter tracking the number of current users of this disk.
Read$Write: States whether the disk is open for read-only, or for both reading and writing.
Open$Connection: A link to the location of the image file on disk.

Operators
Initialize: This zeroes out all entries in the Application and Disk Maps, setting it with no applications or disks open. It then sets up the first entry in the Application Map, as the application entry describing all user-disks.

Application Map Operators:
Search$Appl: This scans the map for a particular application File Number. If found, the entry's offset in the array (open application number) is returned.
Attach$Appl: This copies application information obtained from the Common$Info file, to a free entry in the map, and returns the open application number.

Disk Map Operators:
Search$Disk: This scans the disk map for a particular disk's file-number and returns the open-disk-number (the offset in the map), if it is found.
Attach$Disks: Given a disk (file) number, the quantity of disks requested, and an open application number, this operation obtains entries for the disks and puts the disk's file-number and open-application-number in each disks entry.
Remove$Disks: This deletes the link to the image file and clears the map entry, making it available.
Qty$Open: Returns the number of current users of this disk.
Mark$Open: First the validity of the request is checked. It is invalid if an open for writing is requested, when the disk is already open for read-only, or with Open$Count greater than one. The number of disk opens must also not exceed the maximum allowed for each disk of the application (Max$Users in the application entry). For each disk the Read$Write field is set and Open$Count is incremented. If Open$Count was previously zero (i.e. the disk was previously not in use), then Qty$Open is incremented in the application entry.
Mark$Closed: This marks a disk as closed, and may also have to remove the disk or even its application as well, if they will no longer be in use. The operation is slightly

different depending on whether it is mandatory for the application to have all its disks open at once, or not (i.e. it depends on the value of All$Reqd). The operation can be described as follows:

```
BEGIN:  Mark$Closed
  IF:  (All$Reqd  =  True)
     THEN:
       * Decrement Open$Count, for the disk *
       IF:  (Open$Count  =  0)
         THEN:
           * Decrement Qty$Open for the application *
           IF:  (Qty$Open  =  0)
              THEN:
                * Remove each disk of the application*
                * Remove the Application *
     ELSE:
       * Decrement Open$Count, for the disk *
       IF:  (Open$Count  =  0)
         THEN:
           * Remove$Disk *
           * Decrement Qty$Open for the application *
           IF:  (Qty$Open  =  0)
              THEN:  * Remove the Application *
END:  Mark$Closed
```

Disks$Ready: This checks to see whether the operating system's preparation of files is complete, and if the files are ready for use.

4.3.2.7 Cache

The Cache Resource maintains images of pseudo-disk data in RAM, to speed up response to disk requests from nodes. Its requirements and characteristics were discussed in Section 3.2.6.

The basic unit of data for the cache is a pseudo-disk track, while the basic unit from the view of control is a group of tracks. Therefore the cache has two types of entry. A track entry locates the data block of memory used to store the track and records characteristics of the track, while the track group entry locates a set of track entries and records characteristics applying to the group.

From the view of data organization, the cache links the track groups into a number of lists - one for each open pseudo-disk. It also contains lists controlling the replacement of cache entries and the writing of data to disk.

Data Management

Each open pseudo-disk has a list linking together its track group entries. These disk lists are ordered according to the tracks the groups hold. Each group has a track list, linking the track entries covered by the group, which is also kept in track order.

When a track needs to be fetched to cache, the cache determines the size of the group to be fetched based on the type of disk, which tracks for the disk are already in the cache and the space available in the cache. The track group entry and track entries are prepared first and inserted to the correct position in the pseudo-disk's track group list. The track entries are at this stage marked not present. Track data is read from disk directly to their positions in the cache.

The unit for data update is the sector. Since the cache is track-based, a group of tracks must first be fetched from disk (unless the sector is already in the cache) before the cache can be updated. The sector is then copied from memory to its position in the cache, and record is kept so that it is later written to disk.

Writing Strategy

A write list is maintained, which links groups containing sectors which have been updated. After being updated, a group is given a timestamp and placed at the rear of the list. A group will generally not be written to disk until it has been in the queue for some minimum time - this allows time to receive updates for subsequent sectors in the group. The cache requires writing either when the front group's minimum time in the list has elapsed, the front group is marked as Must$Write, or the list size exceeds a certain value.

The groups are not written to disk in the order of the write list - since these locations may be scattered all over the disk, wasting a lot of disk seek-time. Instead when the front entry is ready for writing, it is used to choose a pseudo-disk to update. This disk's Track Group List is scanned according to track order, and each track containing dirty sectors is selected for writing (Only the dirty sectors within a track will be written to disk). Assuming that the pseudo-disk is stored approximately contiguously, updating a pseudo-disk at a time will require few disk seeks.

Replacement Strategy

All groups which are not currently required for an operation and have not been updated, are linked into a replacement list. When the cache needs entries to store tracks to be fetched from disk, it obtains and clears entries from the front of the replacement list. When a track in a group is needed, the group will be removed from the replacement list and later re-inserted at the rear. This means that the whole group is maintained in the cache, while any member is in use.

Data Description

The cache consists of data entries organized into multiple lists. There are two types of entries, the track entry and the track group entry. Track entries are linked into lists within a group entry, where a group has a list rock pointing to its first track entry. Track group entries have a number of lists running though them - lists for each open pseudo-disk, the write list and the

replacement list. Free track entries and track group entries are also linked into their respective spacelists. The cache contains rocks for each of these lists.

A track entry contains information on the track it represents - such as its location in memory and which sectors have been updated. The group entry can be divided into four parts, containing information on the group, the disk list, writelist and replacement list. It identifies the tracks covered by the group, links the track list to the group, and has pointers to other group entries in the various lists.

Cache data structures and control variables include:

- Disk$Lists$Array: An array the size of the maximum number of simultaneously open disks. Each entry is the rock of a track group list for a particular disk - i.e. it points to the first track group entry in the list.
- Track$Group$Array: A fixed size array providing a pool of track group entries.
- Track$Array: A fixed size array providing a pool of track entries. Each entry in the array is also associated with a track-sized data block to store track data.
- TG$SLR: The Track Group Space List Rock, pointing to the first free track group entry and front entry of the space list.
- Free$Groups: Total group entries in the space list.
- Track$SLR: The Track Space List Rock, pointing to the first free track entry and front entry of the space list.
- Free$Tracks: Total track entries in the space list.
- WL$Front, WL$Rear: A pointer to the track group entry at the front and at the rear of the write list.
- WL$Size: Total entries in the write list.
- RL$Front, RL$Rear: A pointer to the track group entry at the front and at the rear of the replacement list.
- RL$Size: Total entries in the replacement list.

A *Track Entry* contains:

- TL$fp: The Track List forward pointer. It points to the next track entry belonging to the same group, forming the track list of the group. It is also used to link free entries into a space list.
- Track$Pos: The location in memory of the data block assigned to hold the track data.
- Present: This indicates whether the track data has yet been read from disk to its memory data block.
- Sector$Array: An array with an entry for each sector of the track. An entry is set true to indicate that a sector has been updated.

A *Track Group Entry* contains:

Information identifying the group

- Track$A, Track$B: The extents of the group within the pseudo-disk i.e. the numbers of the first and last tracks held by the group.
- Disk: The disk to which the group of tracks belong.
- Qty$Ops: The number of outstanding operations still to be performed on this group.
- Dirty: Indicates whether any of the group's tracks have been updated.

TL$LLR: Each group has a track list, linking the track entries belonging to the group. TL$LLR is the Track List Rock identifying the first track entry in this list.

Information for Disk Track Group Lists

TGL$fp: The Track Group List forward pointer. It Points to another track group entry, forming the disk's track group list. It points to the next track group (in track order) belonging to the same disk. It is also used to link free entries into a space list.

Information for the Replacement list

RL$fp, RL$bp: Forward and backward pointers to track group entries in the replacement list.

Information for the write list

WL$fp, WL$bp: Forward and backward pointers to track-group entries in the write list.

Must$Write: This indicates that the group must be written when it reaches the front of the list, even if the minimum time has not elapsed. Other groups will not be inserted in front of a group marked as Must$Write.

Time$Stamp: A time stamp is given to an entry when it is added to the write list, so that it can be determined how long a group has been ready for writing.

Operators

Initialize: Prepares the cache for operation. The initial position is with all track entries and track group entries in their respective space lists. Each disk list, the write list and the replacement list are empty. Data blocks are created and linked to specific track entries.

Increase$Ops: Increases the count for outstanding operations to be performed on a particular group, and removes the group from the replacement list.

Reduce$Ops: Reduces the count for the outstanding operations for which a group is required. If no operations are outstanding, and the data has not been updated (i.e. there are no dirty sectors), the group is added to the replacement list.

Mark$Present: Marks a track entry to indicate that the track data has been read from disk. This track's data is immediately available for use - the whole track group need not have been read.

Query$Present: Determines whetner a track entry's data is available for use. although an entry may exist in the cache, the track data may not yet have been read from disk.

Fetch: Searches the cache for the requested pseudo-disk track. If necessary, a new entry is inserted into the list, and a message requesting the track data is sent to the Disk Read Queue . The group entry is removed from the replacement list and the quantity of outstanding operations for the group is incremented. If the requested track is the last entry in its track group, then a prefetch operation is performed for the subsequent track. It returns the location in memory of the track data, and identifies the group entry so that later the outstanding operations can be decremented.

Prefetch: The prefetch operation is used to fetch tracks which are expected to be required in the future, but have not been explicitly requested. The operation is the same as the fetch operation except that the request does not count as an outstanding operation (i.e. the prefetched group's quantity of operations is not incremented).

WL$Ready: Examine the front entry in the write list to determine whether a write-to-disk must be performed. This depends on the size of the write list, on the time elapsed since the group was last updated, and whether the group is marked as 'Must$Write' (indicating that its disk is to be removed from the cache).

Next$Write: Once the Disk Write Process has selected a disk to update, the Next$Write operation determines the group and track to be written. The caller specifies the last track written, and the disk's track group list is searched for the next dirty track in sequence. If no previous track is specified, it will search from the start of the list.

Perform-Write: Copies a sector from the sector pool, to its position within the track data of a track entry. The track containing the sector must be in the cache. Single sectors are not stored in the cache. A write operation requires that a track must be fetched to cache and the sector then copied to its position in the track.

Flush: Attempts to remove all entries for a given disk from the cache. If a group contains updated information it cannot remove the entry, but will mark the group for compulsory writing, and insert it at the front of the write list. The operation is successful when there are no entries for the specified disk left in the cache.

4.3.2.8 Disk-Read Queue

This queue contains requests for fetches from disk required by the cache. It allows the decoupling of the cache resource from the disk reading process. The cache message specifies the group of pseudo-disk tracks required, and the positions in cache memory where the data must be stored.

Data Description

Each variable-length message entry contains:

Qty: The quantity of pseudo-disk tracks to be read.

Disk: The open disk number (of an entry in the Opened-Disks-Table) for the pseudo-disk file to be read.

Group: The track-group entry in the cache for which the fetch is required. This is used by the disk-read process to update the cache.

Track$A: The first pseudo-disk track to be read - this provides an offset in the file image of where to start reading.

Track$List: A list of track entries in the cache. These contain the location in memory where the track data must be stored.

Operators

The resource is a standard FIFO queue. Standard enqueue, dequeue and initialization operations are provided.

4.3.2.9 Node Number

A variable has to be passed to each server process when it is initiated, to indicate the number of the node which the process will serve. The number identifies which of the node queues to take requests from, and which node to specify when sending output to the network.

Data Description
There is a single data item. It is used by the queue producing process to transfer the node number to a newly created server process. Each of the server processes reads its node number from the same data item.

Operators
Read and Write operations are available. Synchronization is needed to ensure that a new server process waits until the queue producing process has written to the variable, before attempting to read it. The queue producer must not initiate another server until the variable has been read.

4.4 INTERACTIVE SERVING - 3rd LEVEL DECOMPOSITION

The initialization and node-server processes of the second level, need further division into sub-processes. One such sub-process is common to both - the common-disk opening process. This performs the operations on resources, needed before an applications disk can be attached to provide data.

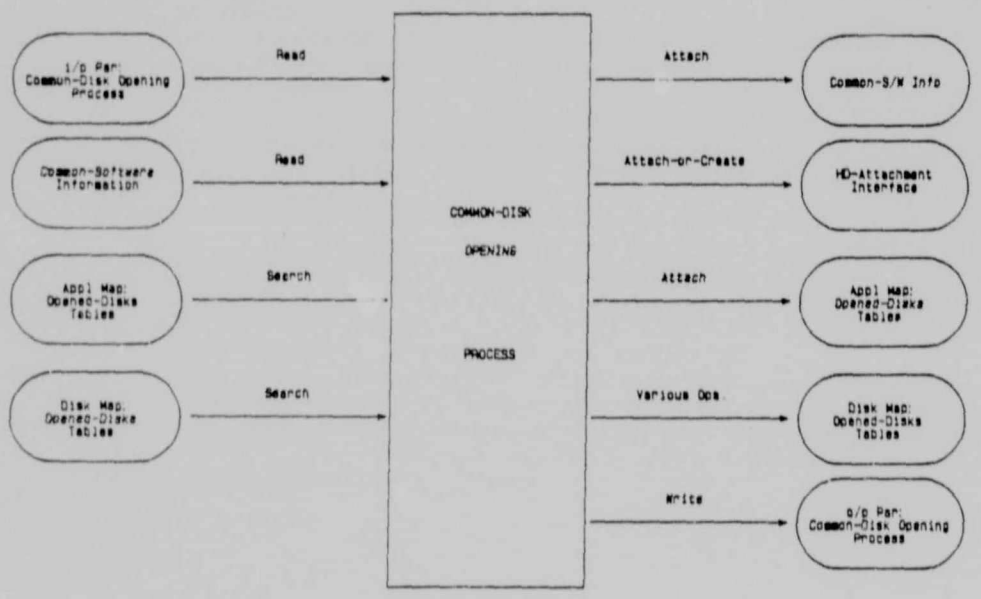
4.4.1 COMMON DISK OPENING PROCESS

Given a file number, this process creates an open link to the pseudo-disk(s) associated with that file number. By 'open', it is meant that all the resource operations which are necessary before data can be obtained from the pseudo-disk, have been done. This involves:

- reading the application's characteristics
- resolving access
- creating links to the file-images of the pseudo-disks (creating the files themselves if they do not exist).
- creating entries in the open disks resource for the application and disks.
- altering existing entries in the open disks resource for the application and disks.

To do this it must perform a number of operations on the HD$Attachment Resource, Common Software Information Resource, and the Open-Disks Tables. Figure 4.4.1 illustrates the interaction with these resources, and the operations performed on them.

FIG 4.4.1 - COMMON-DISK OPENING PROCESS



The process behaviour can be described as follows. At any stage the operation can fail (e.g. access is denied, or the application does not exist), requiring that the resources be restored to their original states. Such complications are not presented. The process differs depending on whether a whole application (all its disks), or a single disk of an application has been requested.

```
PROCESS  Common$Disk$Open
  BEGIN:
    * Receive parameters of the request - includes the file
      number and the callers access characteristics
        Rqst$Disk: File number of disk(s) requested
        Read$Write: The type of access requested
        Reqd$Load: Type of load required (Immediate/Flat)
        Caller$OS: Operating System which caller is using
        Access$Config: User Category of the caller *

    * Extract the application from the file number *
    CASE: * File number - determines qty of disks requested *
      * Request Single Disk - File number is that of a Disk *
        BEGIN:
          * Search for application in Open$Appl$Map *
          IF: * Application is in Map *
            THEN:
              * CALL Query$Access *
              IF: * Access is granted *
                THEN: * CALL Mark$Single$Open *
            ELSE:
              * CALL Get$New$Appl *
              * Attach application to Open$Appl$Map *
              IF: * All$Reqd is set for this Application *
                THEN: * CALL Load$All$Disks *
                ELSE: * CALL Load$Single$Disk *
        END: * Request for a single disk *

      * Request Application - File number is an Application *
        BEGIN:
          * Search for application in Open$Appl$Map *
          IF: * Application is in Map *
            THEN:
              * CALL Query$Access *
              IF: * Access is granted *
                THEN: * CALL Mark$All$Open *
            ELSE:
              * CALL Get$New$Appl *
              * Attach application to Open Appl Map *
              * CALL Load$All$Disks *
        END: * Request for all disks in an application *
      END: * Case *
    * Return output parameters to caller
        Qty: The quantity of disks opened
        First$D$Num: Disk entry of first disk opened. Multi-
          iple disks opened will be in consecutive entries
        Status: Whether the attempt was successful *
  END:  Common$Disk$Open  Process
```

Note: Query$Access, Get$New$Appl, Mark$Single$Open, Mark$All$ Open, Load$Single$Disk and LoadAllDisks, are further sub-processes.

Query$Access determines access to applications by examining the entries in the Access$Bytes string, which refer to the caller's access configuration. The first entry of the pair is for read and the second for write access. The callers Operating System and the type of load requested are also compared with those set for the application.

```
PROCESS  Query$Access
  BEGIN:
    * Compare caller's with application's characteristics *
    IF: * Request's Load$Type and Operating system do not
match
              that in the applications entry *
                        OR
        * User category is not given access in Access$Bytes
            for the type of use (Read$Write) requested *

        THEN: * Access is denied *
        ELSE: * Access is granted *
  END:  Query$Access
```

GetNewAppl reads the application's characteristics from the Common$Info Resource, determines access and creates a link to the application.

```
PROCESS  Get$New$Appl
  BEGIN:
    * Common$Info_Read - Read application's characteristics *
    * CALL Query$Access *
    IF: * Access is granted *
      THEN: * Common$Disk_Attach - create link to application *
  END:  Get$New$Appl
```

Mark$Single$Open marks the required disk as open in the Open Disks Map. It may have to load the disk first, even though the application is already in use.

```
PROCESS  Mark$Single$Open
  BEGIN:
    * Search for Disk number in Open Disk Map *
    IF: * Disk is in Map *
      THEN:
        * Mark Disk open in Open Disk Map for type
            of access requested by Read$Write *
        * Wait until Op$Table_Disks$Ready is true for this
            disk entry i.e. ready for use *
      ELSE:
        * CALL Load$Single$Disk *
  END:  Mark$Single$Open
```

MarkAllOpen marks all disks of the application as open.

```
PROCESS  Mark$All$Open
  BEGIN:
    IF: * All$Reqd is true for this Application *
      THEN:
        * Search for the first Disk number in Open Disk Map *
        * Mark each disk of the application open for type of
            access requested by Read$Write *
        * Wait until Disks$Ready is true i.e. ready for use *
      ELSE:
        * It is not possible to open all disks at once,
            unless all are required to always be open *
  END:  Mark$All$Open
```

Load$Single$Disk attaches an entry for the disk to the Open Disk Map, marks it as open and creates a link to the file associated with the disk.

```
PROCESS  Load$Single$Disk
  BEGIN:
    * Op$Table_Disk$Attach - create entry in the Open Disk
        Map, for this disk *
    * Mark Disk open in Open Disk Map for type of access
        requested by Read$Write *
    * HD$Attach_Open$Create$File - link entry to file *
  END:  Load$Single$Disk
```

LoadAllDisks attaches an entry in the Open Disk Map for each disk of the new application. The required disk is marked as open and a link is created to the files for the disks.

```
PROCESS  Load$All$Disks
  BEGIN:
    * Op$Table_Disk$Attach - create entries for each disk *
    * Op$Table_Mark$Open - Mark each disk requested as open
        for the requested type of access *
    Repeat: * For each disk entry *
        * HD$Attach_Open$Create$File - Create link in entry
            to file *
  END:  Load$All$Disks
```

Author Hildyard Mark William
Name of thesis Design Of A Disk-based File Server Involving The Mapping Of "pseudo-disks". 1989

PUBLISHER:

University of the Witwatersrand, Johannesburg
©2013

LEGAL NOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.