

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/

4.4.2 INITIALIZATION PROCESS: Further Decomposition

Initialization can be separated into two processes:
- Preparation of the File Server: Initializing the server's internal and external resources and its processes.
- Configuring the File Server: Loading data which modifies (and optimizes) the functioning of the file server according to expected conditions or periods of use.

4.4.2.1 Resource Preparation Process (Fig 4.4.2a)

This process performs the initialization operation on all first level resources and those comprising the network interface. It also initiates the queue-producing, disk-read, disk-write and rack-send completion processes. The server processes for each node, however, are initiated from the queue-producing process, when it first receives a request from that node. It has the following characteristic:

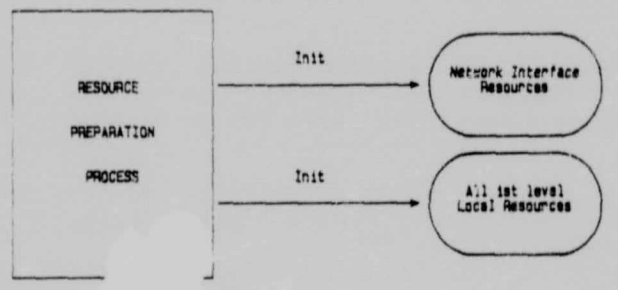
```
PROCESS:  Initialization
  BEGIN:
     * Initialize the 1st level and network-interface
         resources *
     * Perform the Configuration Process *
     * Initiate concurrent processes *
  END:  Initialization
```

4.4.2.2 Configuration Process (Fig 4.4.2b)

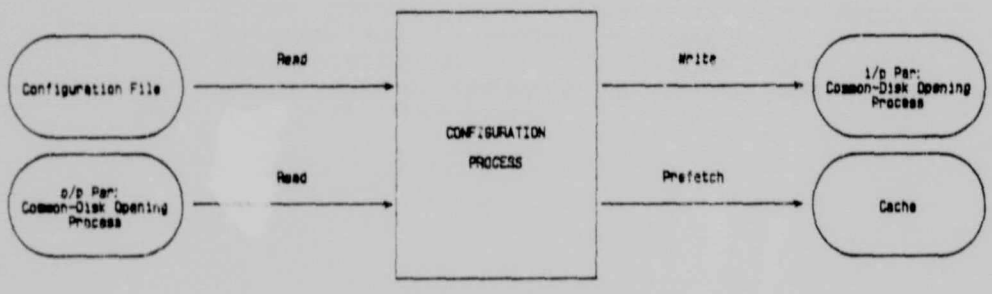
Configuration entails using current selections for any flexible characteristics of the file server. The process has to read the configuration file, and determine settings such as the default operating system. It also reads the ID-numbers of the disks which are most often used. Each of these disks are opened and will remain so for the duration of the session, saving time when a user requests one of these disks. At any stage, the disks are open at least once, so that the disks are never actively cleared from the cache. It has the following characteristic:

```
PROCESS:  Configuration
  BEGIN:
     * Read the default operating system from Config$File *
     * Call the Common$Disk$Open process to open the disks
         of the default operating system *
     * For each disk-ID listed in the Config$File *
        * Read the disk-ID *
        * Call the Common$Disk$Open process *
  END:  Configuration
```

FIG 4.4.2 - DECOMPOSITION OF THE INITIALIZATION PROCESS



(a) RES ATION PROCESS



(b) CONFIGURATION PROCESS

4.4.3 NODE-SERVER PROCESS: Further levels of Decomposition

The file server is not only required to provide disk data, but must provide functions for attaching different pseudo-disks to a node's drives. Therefore the server must maintain an image of the drives for the node which it is serving. It must also keep information on the user currently at its node. Each function provided to the user can be seen as a process at this level.

4.4.3.1 Node-Server Process: ***Sub-Processes***

The individual node-server process can be divided into processes for initialization of the node-server, decoding of requests, and with separate processes to handle requests for each of the functions provided to users.

4.4.3.1.1 *Initialization Process* (Fig 4.4.3a)

The process reads the node number which it must serve, after it has been initiated. It initializes internal resources and connects the default operating system to the drives.

```
PROCESS: Initialize Node-Server
  BEGIN:
     Read (Node$Number)
     * Inform Producer process Node number has been read *
     * Initialize the Drive$Map Resource *
     * Zero$Current$Settings - for Current User Record *
     * Attach default operating system to node's drives *
  END: Initialize Node-Server
```

4.4.3.1.2 *Decoding Process* (Fig 4.4.3b)

Decoding is a continual process reading requests, identifying the function requested, and running the process which handles this function. It has the following characteristic:

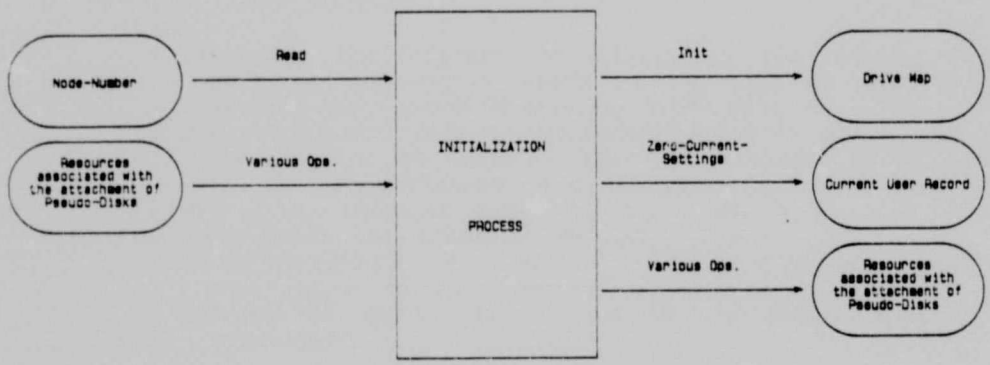
```
PROCESS: Decode Request
  BEGIN:
     Repeat: Forever
       BEGIN:
          Dequeue (Node-Queue)
          * Interpret and process request *
          CASE: * Fn identifies request as *
            Read$Track:              * CALL Read$Track *
            Write$Sector:            * CALL Write$Sect *
            Sign$On:                 * CALL Sign$On *
            Sign$Off:                * CALL Sign$Off *
            Attach$Common$Disk:  * CALL Attach$Common$Disk *
            Attach$User$Disk:    * CALL Attach$User$Disk *
            Detach$Drive:            * CALL Detach$Drive *
            Rqst$$OS:                * CALL Rqst$OS *
          END:
       END:
  END: Decode Request
```

4.4.3.1.3 *Read-Track: Request Handling Process* (Fig 4.4.3c)

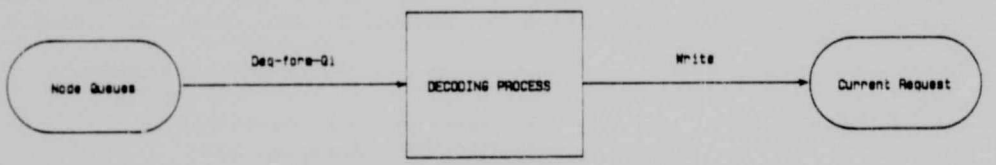
This must interpret and process a request for a track off one of the drives. The process must first identify which pseudo-disk is connected to the drive. It has the following characteristic:

```
PROCESS: Read$Track
  BEGIN:
    * Interpret the request: It identifies the Drive, Head
        and PC-Track to be read at the node *
    * Identify which disk is connected to this drive and
        convert the PC_Track + Head to a single track offset
        in the Pseudo-disk's file image *
    * Fetch the required track to the cache *
    * Wait until the track is in the cache *
    * Enqueue an output message, containing the:
        Function: Same as the request function
        Cache$Group: Track$Group in cache containing this
           track - used to later update the cache.
        Drive, PC-Track, Head: as supplied in the request
        Track$Pos: Location of track data in cache memory*
  END: Read$Track
```

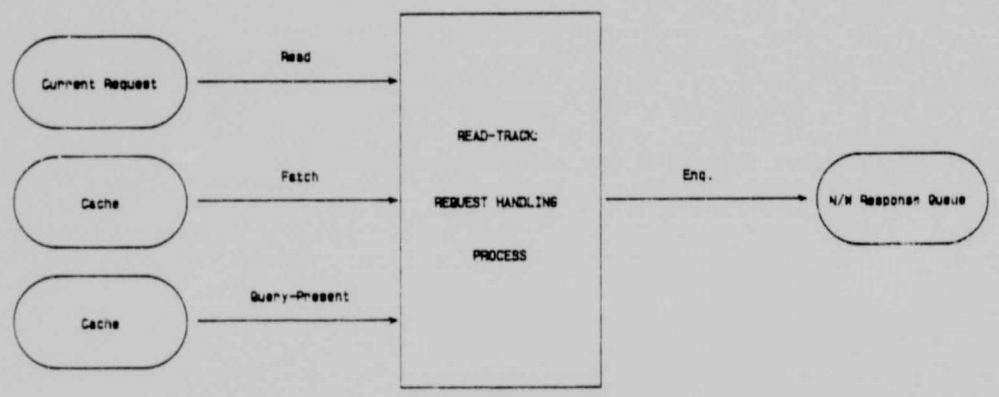
FIG 4.4.3 (a-c) - DECOMPOSITION OF THE NODE-SERVER PROCESS



(a) INITIALIZATION PROCESS



(b) DECODING PROCESS



(c) PROCESS REQUEST TO READ A TRACK

4.4.3.1.4 *Write-Sector: Request Handling Process* (Fig 4.4.3d)

This process handles requests to update a pseudo-disk with sector data. It has the following characteristic:

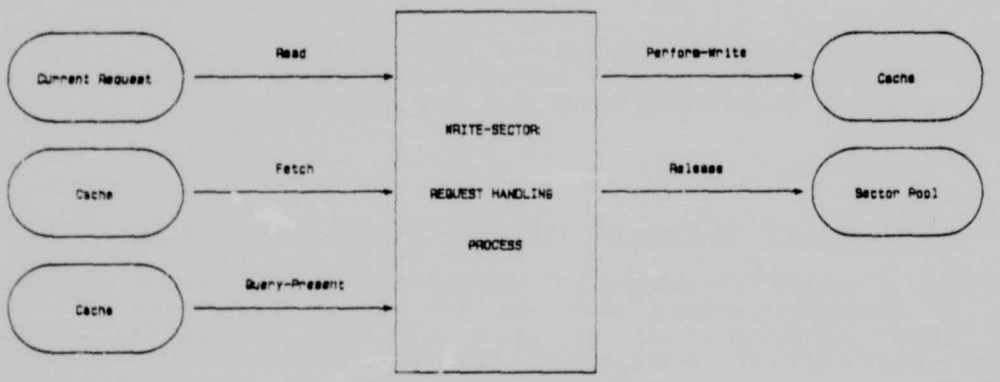
```
PROCESS:  Write$Sect
  BEGIN:
    * Interpret the request: It identifies the Drive, Head,
      PC-Track and sector which was written to, and the
      Sector Pool entry containing the data *
    * Identify which disk is connected to this drive and
      convert the PC_Track + Head to a single track offset
      in the Pseudo-disk's file image *
    * Fetch the required track to the cache *
    * Wait until the track is in the cache *
    * Cache_Perform$Write - copy the data from its sector
      pool entry to its position in the cache *
    * Release the sector entry back to the Sector Pool *
  END:  Write$Sect
```

4.4.3.1.5 *Sign-On: Request Handling Process* (Fig 4.4.3e)

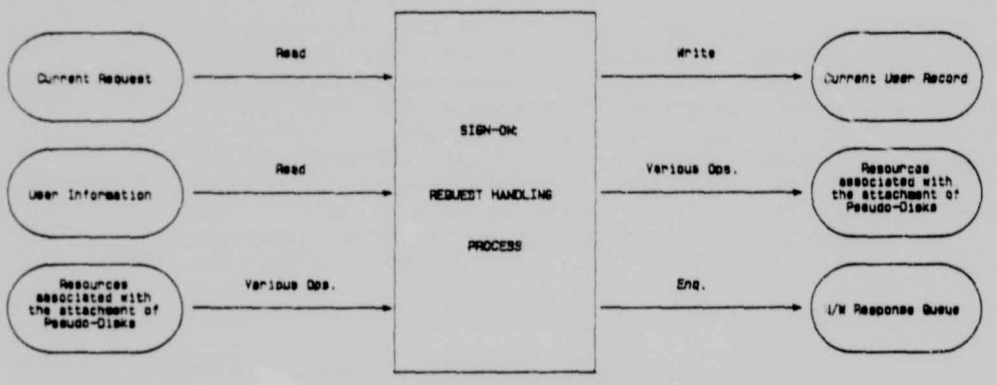
This processes a request to attach a user to the system and allow him access to his disks, and any software to which he is allowed access. It has the following characteristic:

```
PROCESS:  Sign$On
  BEGIN:
    * Interpret the request: It provides the User$Num and
      the password of the user signing on *
    * Read User's characteristics from User$Information
      resource, and verify the password *
    IF: * Successful - i.e. password matches *
      THEN:
        * Copy data from entry in User$Information to
          Current User Record *
        * CALL Attach$OS - attach the Operating System
          specified as Init$OS in User Record *
    * Enqueue an output message, containing the:
      Function: Same as the request function
      Status: Whether request was successful
      User information such as the category of access,
        the operating system and maximum user disks *
  END:  Sign$On
```

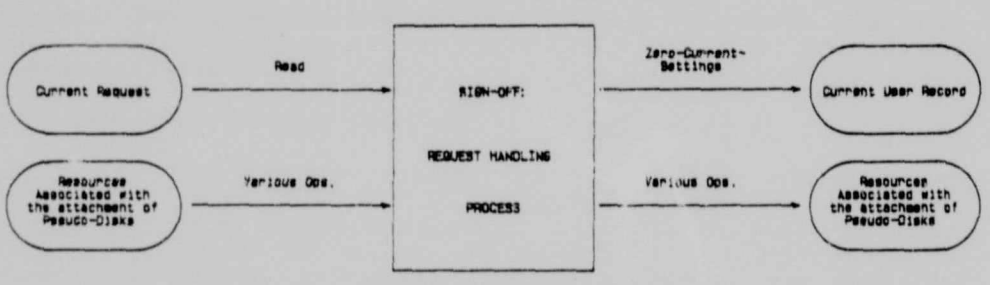
FIG 4.4.3 (d-f) - DECOMPOSITION OF THE NODE-SERVER PROCESS



(d) PROCESS REQUEST TO WRITE A SECTOR



(e) PROCESS REQUEST TO SIGN-ON



(f) PROCESS REQUEST TO SIGN-OFF

4.4.3.1.6 *Sign-Off: Request Handling Process* (Fig 4.4.3f)

The sign off function disconnects a user from the node, which must be reset to its initial state before the user was attached. It has the following characteristic:

```
PROCESS:  Sign$Off
  BEGIN:
    * Remove link in User Record to location of user disks *
    * CALL Zero$Current$Settings - Reset User Record *
    * CALL Attach$OS for the default Operating System *
  END:  Sign$Off
```

4.4.3.1.7 *Attach-Common-Disk: Request Handling Process*

(*cf*: Fig 4.4.3g) The process handles a request to attach a common disk to the node's drives. If the file number requested is an application number, then all disks in the application must be attached. If successful, the disk is attached to the first free drive, or if multiple disks were requested, to sequential drives. It has the following characteristic:

```
PROCESS:  Attach$Common$Disk
  BEGIN:
    * Interpret the request: It contains the File number,
        load type and read/write indicator desired *
    * CALL Common$Disk$Attach - open the disks, verify
        access and attach disks to available drives *
    IF: * Successful AND Load$Type is Immediate *
      THEN: * Prefetch first set of disk's tracks to cache *
    * Enqueue an output message, containing the function,
        the status, the first drive to which a new disk was
        attached, and the quantity of drives attached *
  END:  Attach$Common$Disk
```

4.4.3.1.8 *Attach-User-Disk: Request Handling Process*

(*cf*: Fig 4.4.3h) When a user signs-on, all the user disks are attached to drives. However, disks may be disconnected and required later. The request contains the number of the user disk required, not a disk's file number. If zero, it means that all of the user's disks must be connected. It has the following characteristic:

```
PROCESS:  Attach$User$Disk
  BEGIN:
    * Interpret the request: It contains the number of the
        user disk and the read/write indicator desired *
    * CALL User$Disk$Attach - verify user has the disk(s),
        open the disk(s) and attach to available drives *
    * Enqueue an output message, containing the function,
        the status, the first drive to which a new disk was
        attached, and the quantity of drives attached *
  END:  Attach$User$Disk
```

4.4.3.1.9 *Detach-Drives: Request Handling Process*

(*cf*: Fig 4.4.3i) This processes a request to detach disks from a node's drives. The request specifies the starting drive and the quantity of drives to detach disks from. It has the following characteristic:

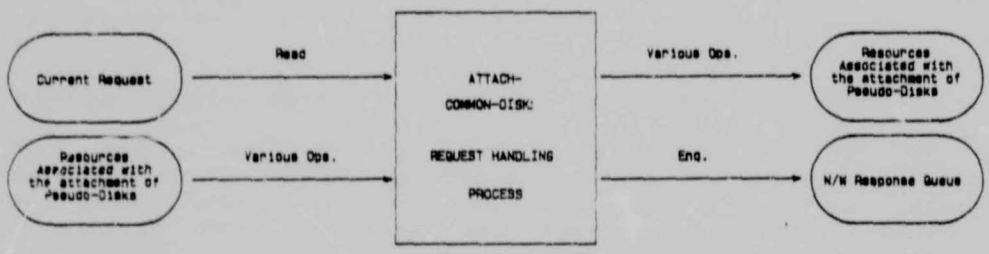
```
PROCESS:  Detach$Drives
  BEGIN:
    * Interpret the request: It contains the first dr  and
      the quantity of drives from which to remove disks *
    * CALL Drive$Map_Detach *
  END:  Detach$Drives
```

4.4.3.1.10 *Change-Operating-System: Request Handling Process*

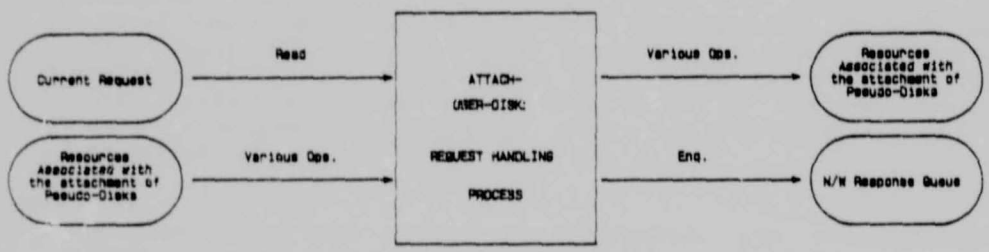
(*cf*: Fig 4.4.3j) The process handles requests for a change in operating system. It has the following characteristic:

```
PROCESS:  Rqst$C
  BEGIN:
    * Interpret the request: It contains the number of the
      operating system desired *
    IF: * The requested operating system is not current *
      THEN:
        * Current.OS$Num := Rqst$OS *
        * CALL Attach$OS - Detaches all drives, attach the
          operating system with a directory disk and user
          disks. If an error occurs, attach default O.S. *
  END:  Rqst$OS
```

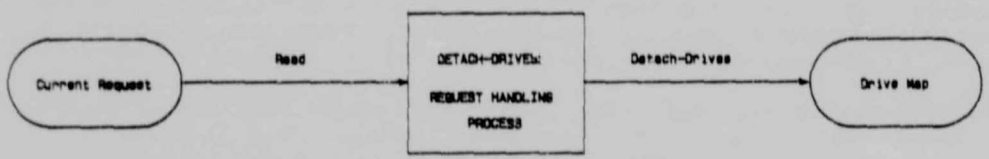
FIG 4.4.3 (g-i) - DECOMPOSITION OF THE NODE-SERVER PROCESS



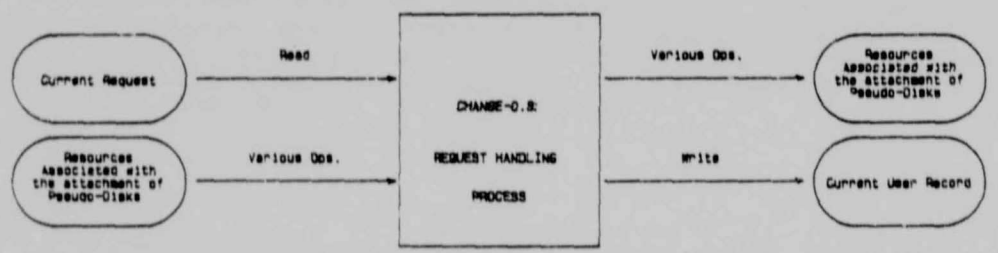
(g) PROCESS A REQUEST FOR A COMMON DISK



(h) PROCESS A REQUEST FOR A USER DISK



(i) PROCESS A REQUEST TO DETACH DRIVES



(i) PROCESS A REQUEST FOR AN OPERATING SYSTEM

4.4.3.2 Node-Server Process: *Local Resources*

The *node-server process* must have local resources which reflect the current status of the link. This includes an image of which disks are connected to the client's drives, the characteristics of the current user, and the current outstanding request.

4.4.3.2.1 *Drive Map*

This resource maps each of a node's disk drives, to the open disk entry of a pseudo-disk currently connected to the drive. There is a similar map maintained at each node's network interface - although here, the drives are mapped to the file numbers of pseudo-disks (*cf*:section 3.3). Clearly, any change in the server's map must be coupled to changes at the node's map.

Data Description

Drive$Array: Consists of an array, with an entry for each drive. Each entry shows the Open Disk Number (in the Open Disks Map) of the pseudo-disk connected to this drive.

Perm$Drive$Limit: Any drives below this limit cannot have their disks detached - the limit must first be lowered. This is used to ensure that there is always an operating system disk connected as a node's first drive.

Operators

Init: This sets each drive as unattached, and the Permanent limit to zero.

Crunch: This is used to tidy up after detachments. Attachments are shifted to different drives until all the drives with attached disks, are the low-numbered drives.

Set$Perm: This sets the Perm$Drive$Limit to the highest-numbered attached drive, making all the currently attached disks 'permanent' - i.e. they cannot be removed via a detach operation.

Attach-Disks: This takes the given open disk entry and quantity required, and locates sufficient free sequential drives - this may require that the Drive$Array be crunched. The open disk number (and sequential disk numbers, for quantities greater than one) are attached to the drive(s).

Detach-Drives: Given a starting drive and a quantity, this will remove disks from the specified drives - however permanent drives will not be removed. It clears the entries in the DriveMap, waits for any disks open for writing to be flushed from the cache, and marks each of the disks closed in the Open Disk Map.

4.4.3.2.2 *Current User Record*

This is a single record used to store a user's characteristics once he has signed-on - information is obtained from the User$Information resource.

Data Description
There is a single record with the following items:
- User$Num: User number of the current user.
- Access$Config: Category of user - used to determine access to applications.
- Max$Qty: The quantity of user disks allowed for this user.
- OS$Num: The current operating system in use, and connected to the bootable drive.
- User$File$Num: File numbers for each user disk are constructed from the user number (*cf*: section 3.2.3 and fig 3.2.3.1).
- Dir$Connect: A link to the location of the user's disks on the server's disk.

Operators
Read and write operations are available. An operator Zero$ Current$Settings resets each item in the record to its default value.

4.4.3.2.3 *Current Request*

This stores the current request while it is being processed. Its structure is that of an unstructured array, since each of the requests has a different structure. The first entry is the request function.

4.4.3.3 Node-Server Process: *Further Sub-Processes*

Further decomposition is required of those request handling processes which involve the attachment of disks. Processes can be isolated which manage operations on resources to open user disks, attach user or common disks, and attach an operating system. (A process to manage opening common disks has already been identified externally to the node server). These form sub-processes of request-oriented processes, which interpret the action required by requests.

The operations can fail at any stage (e.g. if access is denied), requiring that the resources be restored to their original states. Such complications are not presented in the process behaviour descriptions given. The sub-processes are illustrated in figure 4.4.3.3(a-d), and the modified request handling processes in figure 4.4.3.3(e-j).

4.4.3.3.1 *Attach Operating System* (Fig 4.4.3.3a)

This process manages operations required to connect an operating system to a node's drives. It must use the operating system number to obtain the application containing the disks for running the operating system, and attach these disks. If a user is connected, it must also attach a directory disk (for this operating system and the user's category), and the user's disks. It has the following characteristic:

```
PROCESS:  Attach$OS
  BEGIN:
    * Set Perm$Drive$Limit to zero *
    * CALL Drive$Map_Detach - for all drives *
    * Delete link in User Record to user disk locations -
        disks for a different operating system are located
        in a different place *
    * CALL Common$Disk$Attach - for the OS's file number *
    IF: * No User is currently connected *
      THEN: * CALL Drive$Map_Set$Perm *
      ELSE:
        * CALL Common$Disk$Attach for the directory disk *
        * CALL Drive$Map_Set$Perm *
        IF: * Quantity of disks allowed user is not zero *
          THEN:
            * Attach User directory *
            * Attach each user disk for write access *
END:  Attach$OS
```

4.4.3.3.2 *User-Disk Attachment Process* (Fig 4.4.3.3b)

This process manages resource operations required to attach a user disk to the first drive available. It invokes the User$Disk$Open, to first open the disks. It has the following characteristic:

```
PROCESS:  User$Disk$Attach
  BEGIN:
    * Receive the parameters of the request:
        Disk: Indicates which of the user disks is required
        Read$Write: The read/write indicator of the disks *
    * Check that the maximum user disks allowed for the
        current user is not exceeded *
    * Construct a file number from the user number and Disk*
    IF: (Disk = 0)
      THEN: * All user disks for this user will be attached*
      ELSE: * A single user disk will be attached *
    Repeat: * For each disk to be attached *
      BEGIN:
        * CALL User$Disk$Open *
        * CALL Drive$Map_Attach *
      END:
    * Return the first drive connected, the quantity
        connected, and the status *
  END:  User$Disk$Attach
```

4.4.3.3.3 *Common-Disk Attachment Process* (Fig 4.4.3.3c)

This process manages resource operations required to attach a user disk to the first drive available. It invokes the Common$Disk$Open, to first open the disks. It has the following characteristic:

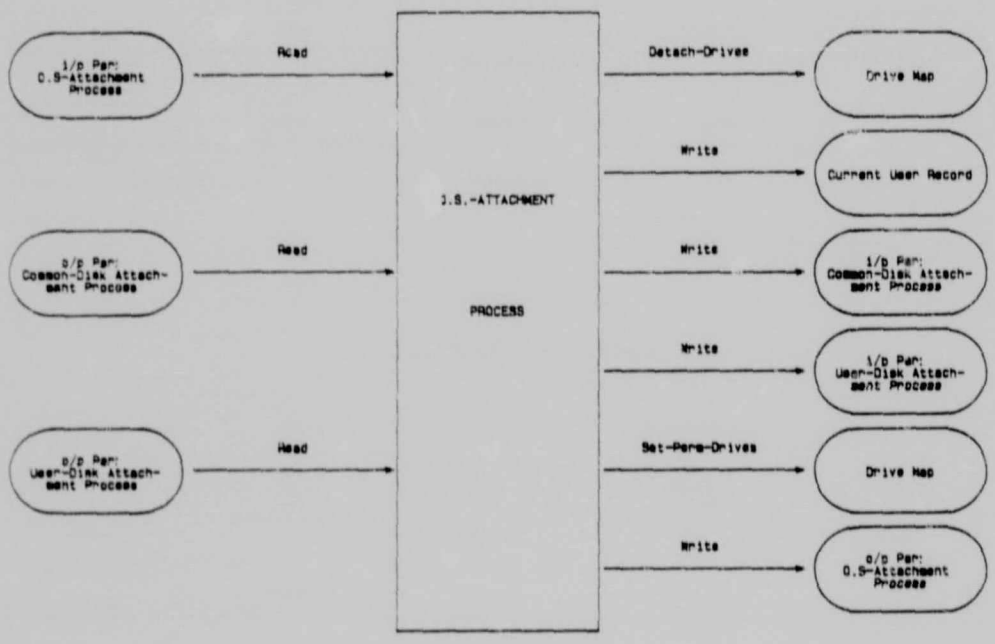
```
PROCESS:  Common$Disk$Attach
  BEGIN:
    * Receive the parameters of the request:
        File$Num: File number of disks required
        Load$Type: Type of load required (Immediate/Flat)
        Read$Write: The read/write indicator of the disks
        OS$Num: The operating system of the caller
        Access$Config: The caller's category of access *
    * CALL Common$Disk$Open *
    IF: * Successful *
      THEN:
        * CALL Drive$Map_Attach *
    * Return output parameters: the first drive connected,
        the quantity connected, and the status *
  END:  Common$Disk$Attach
```

4.4.3.3.4 *User-Disk Opening Process* (Fig 4.4.3.3d)

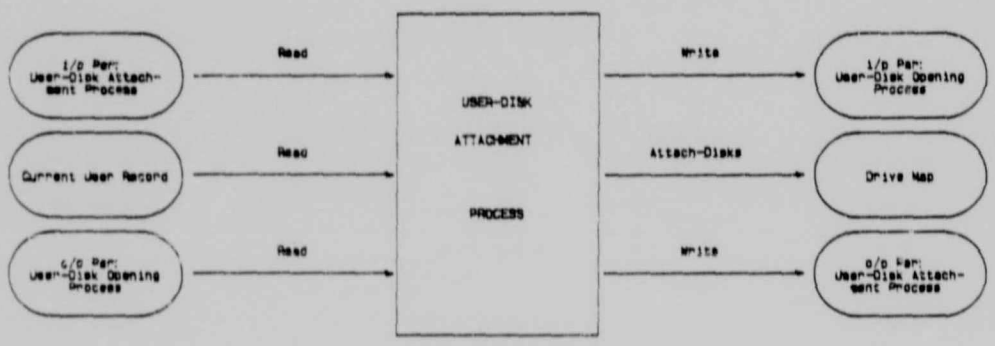
This process manages resource operations required to open a user disk - making it available to be connected to a drive. It has the following characteristic:

```
PROCESS:  User$Disk$Open
  BEGIN:
    * Receive the parameters of the request:
        File$Num: File number of the user disks required
        Read$Write: The read/write indicator of the disks *
    * Search for File number in Open Disk Map *
    IF * Disk is in Map *
      THEN:
        * Mark Disk open in Open Disk Map for the type of
            access requested by Read$Write *
        * Wait until Op$Table_Disks$Ready is true for this
            disk entry i.e. ready for use *
      ELSE:
        * CALL Op$Table_Disk$Attach - create entry in the
            Open Disk Map, for this disk *
        * Mark disk open in the Open Disk Map for the type
            of access required by Read$Write *
        * HD$Attach_Open$Create$File - link entry to file *
    * Return the first drive connected, the quantity
      connected, and the status *
  END:  User$Disk$Open
```

FIG 4.4.3.3 (a,b) - FURTHER DECOMPOSITION: NODE-SERVER PROCESS

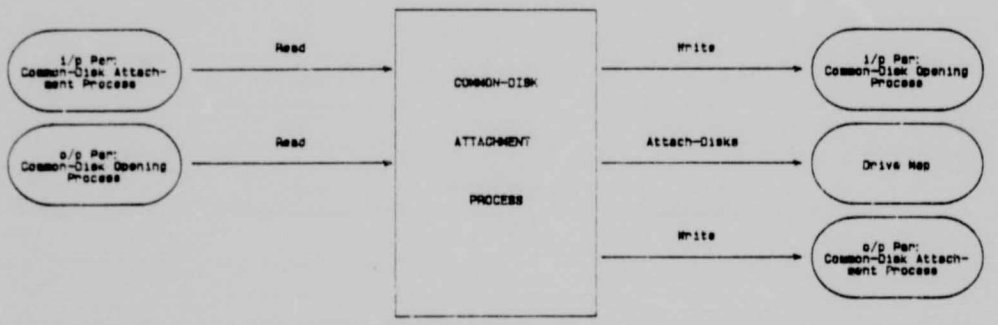


(a) O.S-ATTACHMENT PROCESS

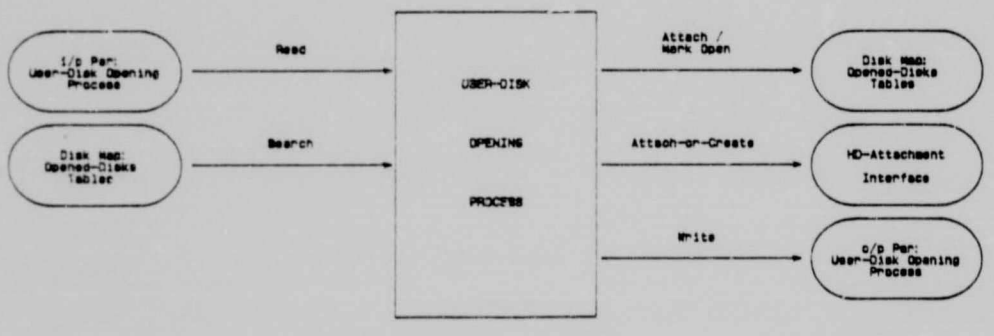


(b) USER-DISK ATTACHMENT PROCESS

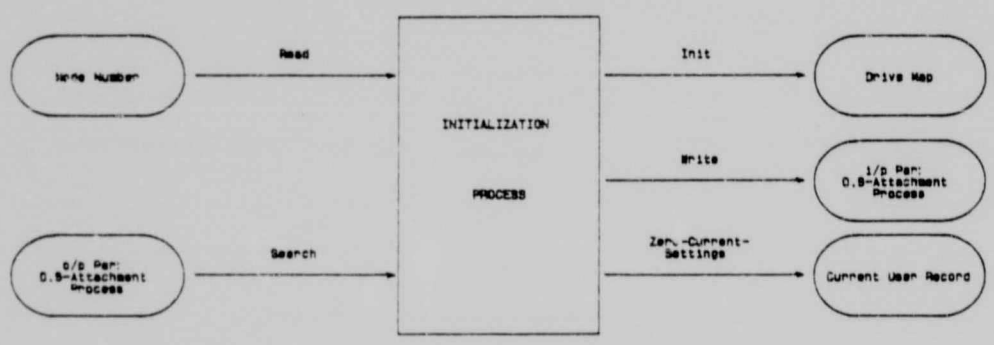
FIG 4.4.3.3 (c-e) - FURTHER DECOMPOSITION: NODE-SERVER PROCESS



(c) COMMON-DISK ATTACHMENT PROCESS

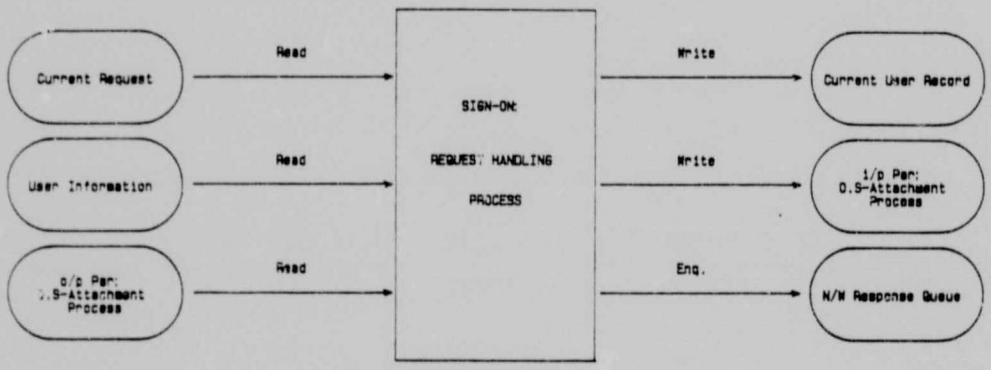


(d) USER-DISK OPENING PROCESS

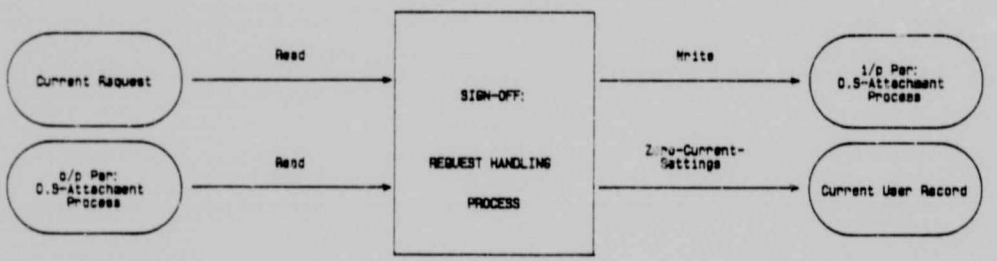


(e) INITIALIZATION PROCESS

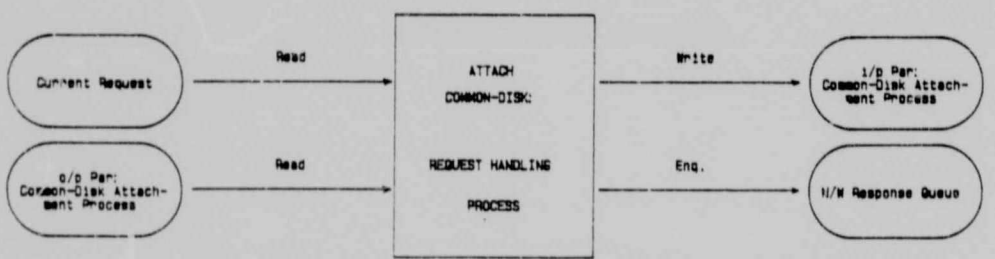
FIG 4.4.3.3 (f-h) - FURTHER DECOMPOSITION: NODE-SERVER PROCESS



(f) PROCESS A REQUEST TO SIGN-ON

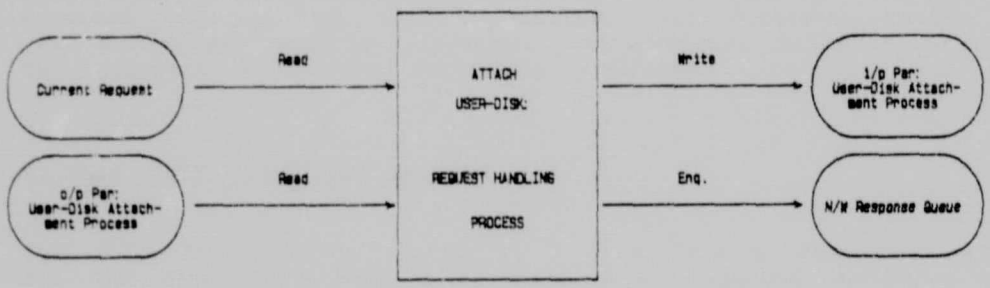


(g) PROCESS REQUEST TO SIGN-OFF

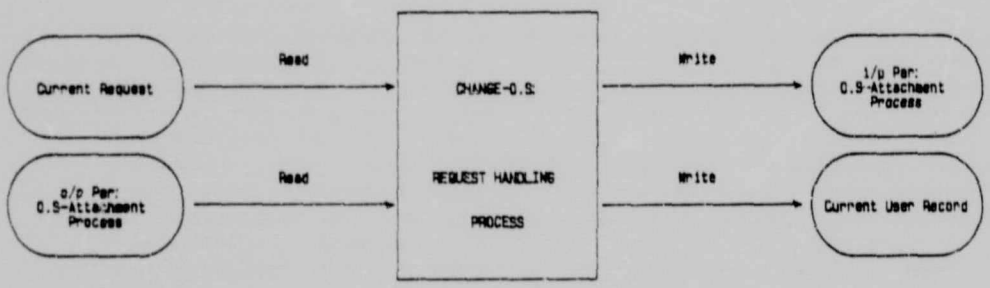


(h) PROCESS A REQUEST FOR A COMMON DISK

FIG 4.4.3.3 (i.i) - FURTHER DECOMPOSITION: NODE-SERVER PROCESS



(i) PROCESS A REQUEST FOR A USER DISK



(j) PROCESS A REQUEST FOR AN OPERATING SYSTEM

5 IMPLEMENTATION OF THE MINIMAL FILE SERVER

The computer chosen for preliminary implementation of the file server was an Intel System 86/330A. The disk capacity of 30 Mbytes and RAM capacity of about 260 Kbytes after the operating system has been loaded are inadequate for suggested disk storage and cache requirements. However, it was chosen on the basis of its availability and that of a multi-tasking operating system, the iRMX-86. Multi-tasking facilitates exploiting the natural concurrency of the application, where a virtual serve is provided to each node in the network. The processes isolated in the design are readily performed concurrently, without the need to implement a complex scheduling algorithm.

5.1 THE iRMX OPERATING SYSTEM

The iRMX 86 operating system is a multi-tasking real-time system for the Intel iAPX 8086 microprocessor - enhanced versions exist for later Intel chips such as the iRMX 286R for the iAPX 80286 [19, pp 390].

A brief overview of the operating system is given here, with particular attention to the features used in the file server. A comprehensive overview is given in the Intel Introduction to the iRMX operating system [20], and [Tucker, 19].

The iRMX 86 is designed as a modular operating system, with a layered architecture, allowing it to be configured to contain just the sub-systems required by the application. It consists of the following sub-systems - all except the nucleus are optional:

- *The Nucleus* is the heart of the operating system providing support for multi-tasking, multi-programming, mutual exclusion, synchronization and inter-task communication.
- *The Basic I/O System* provides file management and the device-independent interface to input and output devices.
- *The Extended I/O System* is an extension to the basic I/O system providing a simpler, more easily used interface to I/O devices.
- *The Human Interface* provides an interface with a terminal-based user, and support for an extensive range of commands.
- *The Terminal Handler* allows a terminal to be used without using the I/O system or Human interface.
- *The Application Loader* allows an application to load programs and overlays from disk.

The operating system has an object-oriented design [19, pp 387]. A user creates and manipulates objects, referencing them via a 16-bit token, allocated by the system. The nucleus supports seven objects - tasks, jobs, segments, mailboxes, semaphores, regions and extensions. The following sections cover each of these objects and disk input/output features.

5.1.1 Jobs

Jobs are the environments in which tasks operate. They provide support for multi-programming - the ability to run in isolation, more than one application simultaneously. Tasks in an application are grouped into a job where they are protected from tasks in other jobs. Each job has its own memory pool from which memory is allocated when its tasks create objects. Each job maintains an object directory which can be used to provide 'run-time binding' - data shared between tasks in different jobs, through a task cataloguing an object under a textual name which is known by the other task. The other task can obtain the token for the object from the job's object directory, using this textual name.

5.1.2 Tasks

Tasks are the active objects in a system. They are executed concurrently, but only one task will actually be running at any time.

A task is always in one of five execution states - asleep, suspended, asleep-suspended, ready, and running.

- A task is *asleep* if it is waiting for a request (for an operating system function) to be satisfied. Most requests allow a task to specify the length of time which it is prepared to wait. A task can also put itself to sleep for a specified time-length.
- A task can be placed in the *suspended* state by another task or by suspending itself. Each task has a suspension depth reflecting the number of 'suspends' outstanding against it. The resume operation reduces this suspension depth. A task leaves the suspended state once it has a zero suspension depth.
- The *asleep-suspended* state occurs when a sleeping task is suspended. It will leave this state either when its sleeping time expires, or another task resumes it.
- A task is *ready* if it is not asleep, suspended or asleep-suspended.
- Only one task can be in the *running* state. This will always be the task in the ready state which has the highest priority.

A task has a priority associated with it - an integer value between 0 (High) and 255 (Low). The priority is used in the task queues of objects, and for the scheduling of tasks.

The scheduling algorithm is priority-based and pre-emptive. This means that the system always executes the highest priority task which is ready to run. If another task with a higher priority becomes ready, it pre-empts the current task.

5.1.3 Segments

Occasionally a task needs additional memory. The nucleus provides calls for allocating and de-allocating memory in objects called segments. A segment is a contiguous block of memory dynamically allocated from a job's memory pool. The token assigned to a segment is the base address of that segment.

5.1.4 Mailboxes

A mailbox is an object which provides for message queueing. It is used to pass a token for an object from one task to another. There are two queues, one for tasks waiting to receive objects, and the other for the object tokens sent. Object queues are processed first-in/first-out, while the task queue can be either FIFO or priority-based - specified when the mailbox is created. A mailbox has a high-performance fixed-size object queue whose memory space is permanently allocated when the mailbox is created. Once its size is exceeded the nucleus must allocate more space, thereby slowing queueing operations.

5.1.5 Semaphores

Semaphores enable tasks to send signals to other tasks. A semaphore is a custodian of abstract 'units'. Tasks request or release a specific number of units. The semaphore allocates units according to how many it has in its custody. It has a queue for tasks waiting to be allocated units. This task queue can be either FIFO or priority-based - determined when the semaphore is created. The semaphore can be used to implement inter-task synchronization and mutual exclusion.

5.1.6 Regions

A Region is an object that guards a specific collection of shared data. Each task awaits its turn at accessing the region. Once it gains access, the task cannot be suspended or deleted by other tasks, until it surrenders access. The Priority of the task that has access, is raised temporarily if another higher priority task is requesting access. Nesting of regions (i.e. a task accessing two regions simultaneously) can cause deadlock, where one or more tasks permanently lock each other out of required resources [21, pp9-5].

5.1.7 Extension Objects

It is possible to create customized objects which have the same protection and control provided to iRMX standard objects. These objects are called extension objects. Any instance of an extension object is called a composite object.

5.1.8 I/O Features

The Basic I/O system provides the user with device-independent I/O. It has two layers. The device driver provides the interface between the hardware controller and the I/O system - there is a device driver for each peripheral in the system. The file driver provides the interface between user tasks and the I/O system, through the provision of system calls. It supports two file interfaces. The physical file interface is used on devices such as printers and terminals which do not have a file structure, although it can be used on disk devices as well. The named file interface provides file organization on mass-storage devices.

Hierarchical file structuring is supported. This means that any number of levels of directories are allowed, where a directory contains files and/or further sub-directories. This aids choosing unique file names, and provides quick-access, logical filing.

File access control is also provided. The job in which a file is created owns the file. This job can specify other jobs which may access this file, and exactly what type of access (read/write) they are allowed.

Control over file *fragmentation* is also provided for, by allowing each file to have its own granularity. Space on the storage device is allocated to this file according to this granule size. This allows the user to trade off speed versus wasted device space, both of which increase with larger granularity.

As mentioned, the iRMX operating system has two input/output sub-systems, where the extended I/O system provides a simpler interface with synchronous I/O, automatic buffering, and the overlapping of I/O operations. It can perform read-ahead and write-behind operations using its buffers, on the assumption that mostly sequential I/O operations are required. This is not efficient for random-access I/O operations.

The Basic I/O system provides more flexibility. It allows a customized buffering technique to be implemented. It also requires synchronization between I/O and processing, allowing one explicit control over overlapping I/O operations.

5.2 CODING PHILOSOPHY

The PL/M programming language was used due to its availability on the system. PL/M is a high-level, block-structured language supported on Intel products. However, it is confined to the Intel Corporation, which is an important concern for portability and maintainability. Since final deployment of the file server is unlikely to be on this machine and may not be in PL/M, the code has been purposely structured to resemble PASCAL, to aid understanding and translation.

Some points on PL/M, need to be high-lighted [22].

- The language is weakly typed, the five basic types being byte, word, double-word, integer and real. Arrays and structured types are supported, although structures within structures are not.
- Character strings and their manipulation are not directly supported.
- Variable parameters are supported by passing a pointer to the variable to the procedure, although abstraction of purpose is not achieved.
- There is no equivalent of the Pascal Repeat-Until statement and arrays are all subscripted from zero, which can lead to logical difficulties and some inflexibility.

While implicit initialization of variables is supported, explicit initialization has been adhered to, since the feature is not always available in other languages.

The coding then, in addition to being an implementation of the design, also complements the design description of chapter 4. It can be used as low-level program description code, for re-implementation in another language or system. The listings also provide the detailed data descriptions for resources, which were not included in chapter 4, to avoid duplication.

5.3 CODING THE DESIGN

The structure of the design, in terms of processes and resources has been fully adhered to. Concurrent processes have been implemented as *tasks*, while the mutual exclusivity of resource access has been attained through the use of *regions*.

Appendix B contains the coding listings. The code for the file server is distributed over a number of files. Rather than presenting a single listing of the entire program with all of its inclusions, each file is presented separately in the appendix, and ordered to correspond with the description of the design in chapter 4.

The file server 'main code' (listing in Appendix B.2.1) combines all of these parts into a single program, through statements to include the files for external resources (the Network-server interface), internal resources, and processes. Execution merely involves invoking the initialization process - which will initialize resources and initiate the other processes.

The program also includes files for global literal definitions, the external procedure definitions of operating system calls used, and a set of simple string functions. The string functions required for comparison (e.g. of passwords) and concatenation (e.g. to construct file names), are not available in PL/M.

5.3.1 Processes

The concurrent processes of the design - processes for queue producing, disk-reading, disk-writing, completing-sends, and the node servers - are implemented as tasks coded as PL/M procedures. They are initiated as tasks by the initialization procedure, which is called by the file server main code. The node-server tasks are created by the queue-producing task, when it receives requests from a node for which a server does not yet exist.

When tasks are created, they are assigned a stack segment, data segment and a pointer to the first instruction of the new task. In this sense they become running instances of a particular sequence of code. This allows multiple tasks to concurrently execute the same code, but with separate data and stack areas, This feature has been utilized in the code for the node-server processes, which therefore has to be re-entrant.

Sub-processes within the above processes, have also been coded as procedures. They are activated by ordinary procedure calls from tasks. The disk-open process is a sub-process to both the node-server and initialization processes (listing in Appendix B.5). Node-server sub-processes are grouped into 'request' procedures (3rd -level, *cf* Appendix B.4.3.4), which interpret requests for each available function, and the 'serving' procedures (4th-level, *cf* Appendix B.4.3.3), which perform actions required by the 3rd-level functions.

5.3.2 Scheduling of Tasks

A simple strategy for the scheduling of tasks was taken as a first step. It was based on the disk being the critical resource and limiting performance of the file server. The Disk-Read Process was given the highest priority, and will take control whenever it is not waiting for the disk or for a read request. Even at full load, it should spend sufficient time waiting for disk accesses, allowing the other tasks take control.

The other tasks (Disk-Write, Queue-Producer, Complete-Sends and all Server tasks) were given equal priority. A running task loses control when it has to queue to access a resource. It will also yield control when it must wait for something to happen to a resource before continuing (e.g. A server task must wait if its node's queue has no entries, or for the cache to have certain tracks marked present by the disk-read process). In this case it yields control by sleeping for a zero time length, which has the effect of placing the task at the rear of the queue of tasks which are ready to run.

This use of the sleep state gives these tasks a polling type characteristic. When idle, they do not wait until an event has happened, but continually become active to check whether an event has happened.

5.3.3 Resources

Resources have been coded as collections of data structures, and the procedures which operate on these data structures. The requirement, that processes only affect the resource data structures through the operations provided, has been adhered to in the code.

Static allocation of memory for resources was preferred. However, large data blocks in the cache and sector pool, were allocated dynamically with requests for iRMX segments - this was done at the initialization of the resources. Dynamic allocation was also necessary to obtain segments which could be used to send disk read requests via a mailbox.

Regions have been used to achieve mutual exclusion, for resources used by more than one task. A process must receive access to the region before performing operations on the resource, after which it must release this region. This leads to two restrictions which were considered when coding.

- Each task must release a region as soon as possible, since other processes may be awaiting access to a resource. This especially means that while accessing a resource, disk operations must be avoided.
- When a task requires access to more than one region simultaneously, it must request access in a specific order and release the region in reverse order. If this convention is not adhered to, such nested access can lead to a deadlock problem - where two tasks each require the region to which the other task has access before they can proceed.

Semaphores which can also implement mutual exclusion, were not used because of the following limitations.

- Low priority tasks which currently have control of the semaphore, do not have their priorities raised if a higher priority task is waiting for the semaphore.
- A task can be suspended even while it is accessing a semaphore.

This means that high priority tasks (e.g. the Disk-Read task) can be delayed indefinitely, while a lower priority task has control of the semaphore. These limitations do not apply to regions [21, pp9-3].

In the Common Disk Open process, it was necessary to nest access's to regions. First the Common Information Resource must be accessed, and then the Open Tables Resource, so that the Common Information file is not used again, before the application is attached to the Application Map. Since this is the only instance of nested regions, deadlock will not occur (cf:21, pp9-5, for further discussion of the deadlock of nested regions). However, any further development requiring nested regions, must maintain this order of accessing i.e. the Common Software Information Resource before the Open Tables Resource.

5.3.4 Queue Resources

Extensive use of mailboxes could not be made, since the queues of the external resources need to be accessed by the network process which runs on a different processor and not under the iRMX operating system. These queues were constructed with a fixed pool-size in static memory.

The Disk-Read Queue was fully implemented using an iRMX mailbox and the mailbox operations of Send$Message and Receive$Message. A segment has to be dynamically created for each request sent from the cache to the Disk-Read Process.

5.3.5 External Resources

The external resources comprising the network interface are shared between two processors - that of the file server computer, and the network interface unit which performs the network process. The shared data structures are located in the file server's memory, to which the network processor has direct access. The two processes must have their own set of operators which they employ to operate on the data structures. Operators required by the network process have been included in listings for the sake of completeness.

The external resources also have regions guarding the exclusivity of the resource from multiple server tasks. However, it was not possible to use regions to achieve mutually exclusive access between the file server and network processes - the network process runs on its own processor, and not under the iRMX operating system. Instead, a system of three semaphores was used, whose locations in the file server's memory are known by the network processor.

Mutual exclusion was applied only to the accessing of critical control variables in the resource - in a multi-processing environment, it would be inefficient to exclude access to the whole resource [18]. The operators used by the file server and the network processes must gain access through the semaphore, before they update the resource's control variables. These operators therefore include the routines to gain access through the semaphore.

The file server and network processor each have two access-related routines. One routine requests access for the process and the other releases the semaphore. There are three semaphore variables. One indicates whether the file server has or is requesting access, and another whether the network has or is requesting access. The third indicates whether the resource is currently being accessed.

There is a set of these semaphores and semaphore access routines, for the Network Input Queue, Network Output Queue and the Sector Pool (*cf*: Appendices B.1.1, B.1.2, and B.1.3). Semaphore access routines required by the network process have been included in listings for the sake of completeness. The mechanics of obtaining access can be described in PDL as follows:

```
VAR * Access Semaphores *
  S1  Boolean  * 1st  Process  has  or  is  requesting  access  *
  S2  Boolean  * 2nd  Process  has  or  is  requesting  access  *
  S3  Boolean  * Access  to  the  resource  has  been  granted
                                  to  one  of  the  Processors  *

PROCEDURE:  Process-1$Access
  * This process is given priority - normally the consumer *
  BEGIN:
    IF * 2nd Process is not requesting access i.e. S2 = False *
       THEN:
         * 1st process requests access - S1 := True *
         REPEAT:
           * Wait until the 2nd processes request is decided *
         UNTIL: * S2 is false OR S3 is true - 2nd process
                      has backed off or has gained access *

         IF * 2nd process has since gained access i.e. S2 = true *
            THEN: * The 2nd process has access - S1 := False *
            ELSE: * Grant access - S3 := True *
    * Return the value of S1 - True if access is granted *
  END:  Process-1$Access

PROCEDURE:  Process-1$Release
  BEGIN:
    S3  :=  False
    S1  :=  False
  END:  Process-1$Release

PROCEDURE:  Process-2$Access
  * Normally the producing process *
  BEGIN:
    IF * 1st process is not requesting access i.e. S1 = False *
       THEN:
         * 2nd process requests access - S2 := True *
         IF * 1st process has since requested access i.e. S1=True *
            THEN * Back-off to allow 1st access - S2 := False *
            ELSE * Grant access - S3 := True;
    * Return the value of S2 - True if access is granted *
  END:  Process-2$Access

PROCEDURE:  Process-2$Release
  BEGIN:
    S3  =  False
    S2  =  False
  END:  Process-2$Release
```

Each process has its own exclusive semaphore which indicates that it has requested access, but the third semaphore indicates that access has been granted. Three semaphores are necessary, to take into account the complete asynchronousity of the two processes. One process may advance any number of steps while the other advances just one. The above algorithm deals with the complication that a process may request access, check that the other process is not requesting access, but by the time it has set its semaphore the other process has since requested access, or may even have received access.

5.3.6 I/O Operating System Features

The Extended I/O system (EIOS) was used both for disk operations involving pseudo-disk files and for the files containing data used by the system. All system calls are therefore synchronous, and tasks wait for completion of a disk operation before requesting another (i.e. disk operations were not overlapped).

Buffering was not used on pseudo-disk file operations. Instead, tracks are read from the files directly to the required place in the cache. This avoids the inefficiency of copying long blocks of data from buffers to the cache, after they have been read from disk. The cache performs its own prefetching strategy. Buffering was used for the data files of the user- and common-disk information, and configuration resources.

File access protection, provided by the operating system, has not been utilized, since it strictly controls access of different jobs to a file. The file server is a single job and all its tasks have access to its files. Access and protection features of pseudo disks are provided at the application level.

The connections to file and directory locations, required in the Open Disk Map and Open Application Map (of the Open Tables Resource) and the Current User Record, are provided via tokens. The textual names of locations can be discarded after making these connections.

The granularity of pseudo-disk files was added as an extra field in the characteristics of an application, allowing optimum granularity to be chosen for each. (It should be a multiple of two (PC) floppy disk tracks - i.e. 9 Kbytes - *cf*: Sect 5.3.8).

5.3.7 Use of Other iRMX Features

The file server was treated as a single *job* in the iRMX system, containing all available memory. It sets itself up as a job, in the 'main' code prior to calling the initialization process.

An iRMX semaphore was used for *synchronization* when passing parameters to a newly created node-server task. The semaphore allows the task to access a common variable of the queue-producing task, to determine which node-number it is to serve.

5.3.8 Alterations to the Design

An alteration was necessary in implementing the cache, due to a discrepancy in the sector-size of the server's disk, and of client floppy disks. At the client , (in MSDOS), the sector -size is 512 bytes, while for the servers disk it is 1024 bytes. It is therefore impossible to read a single track (or an odd number of tracks), or to write a single sector of the pseudo-disk, to the server's disk efficiently. The basic data unit of the cache has therefore been changed to a double-track, which is a whole number of sectors on the server's disk. When writing from cache, an even number of sectors are written, which coincide with the server disk's sectors.

5.4 FUTURE WORK

The file server was never fully implemented, since during the project, the physical interface of the network on which the server is based, was abandoned. This floppy disk drive interface was the entire motivation for the architecture of a file server based on pseudo-disks. The resulting limitations due to floppy disk size and requests for entire tracks (*cf*: Chapter 2), influenced the File Server's design.

With the limitations/restrictions imposed by the floppy disk based interface removed, the file server's design could be revamped. However (as argued in chapter 7), there might still be merit in retaining the underlying attachment/detachment mechanism required for floppy disks.

If the server was further implemented, either as is or with changes, the following areas would need attention.

A *major problem* was encountered with the implementation in PL/M of multiple node-server processes. Such code is required to be re-enterable, for multiple tasks to use the same code segment. This entails that storage for variables is allocated on the task's stack, and not in a static data segment. However, procedures declared to be 'REENTRANT', are not allowed to contain nested procedures [22, pp 8-10]. Code for the serving process, contains many levels of nested procedures, and it could not be re-written as a single, very long procedure.

A solution would be to include in the server program, multiple copies of the node-server process, with different names. A single task would be created for each copy of the code. However, such a solution involves including about 40 copies of this process - which would far exceed currently available memory. It could nevertheless be used as a temporary solution, by including a few copies of the process, to allow testing of the file server. Ultimately, the system would have to be re-coded in a language which fully supports re-entrancy.

Scheduling of tasks could be improved. Scheduling could be made more efficient if its polling characteristic were transformed into an interrupt characteristic, where a task waiting an event suspends itself until it is resumed by another task. Although disk reading is of higher priority than that of writing, once the Disk-Write process has control it should be allowed to perform a number of writes to a pseudo-disk before the disk head is allowed to move elsewhere. A *semaphore* could be used in disk operations to achieve this. Another possible extension is to allow task priorities to be flexible. The write task priority could be increased when the length of the write list in the cache gets too large, to allow it to deal with many write operations.

Only the interactive serving process of the minimal file server has been coded. The project never reached the testing phase, so extensive testing is still necessary and this requires the existence of the data files for applications and users. Off-line management functions through which these files can be created and modified must still be written - essentially this involves standard database editing features. Integrated testing within the network can then be performed and performance evaluated - however, actual performance measurements will require some interpretation, since file server performance is heavily reliant on caching (*cf*:section 2.3.3), while the cache size is nominal.

6 POSSIBLE EXTENSIONS TO THE MINIMAL FILE SERVER

The central function of the file server is the exchange of tracks and sectors between clients and the pseudo-disk files on the server's disk. As such, the minimal functions required of the file server have been identified as those involved with exchange of disk data, establishing user connections and the swapping of disks attached to drives.

However, other issues remain, such as the provision of printing services, the collection of student files for marking by the analysis machine and a condense-and-compact function for eliminating fragmentation within the pseudo-disks. In addition, for the server to be used in a more universal context outside the specialized education environment, it would be desirable that it can be extended to support a variety of functions associated with high level servers (cf Section 1.3).

This chapter attempts to establish how these functions could be implemented, and if they are feasible. In all of this it is important that the efficiency of data transfer is not effected through these functions overloading the capabilities of the server machine.

6.1 COMPACTION AND THE CONTIGUITY OF 'DISKS'

The implications for performance due to fragmentation within the pseudo-disks was discussed in section 2.3, and the need for a condense facility to remove this fragmentation was established. At the same time the pseudo-disk files can be compacted so that they contain only that part of the disk which contains data.

Compaction involves reorganizing sectors within the pseudo-disk, and altering file structure information (e.g. directory information), to reflect the new order. The compaction algorithm is therefore heavily dependent on knowledge of the file system of an operating system, so that this facility should only be provided for operating systems which would be widely used on the system.

In MS-DOS this involves finding each file on the pseudo-disk, and choosing new contiguous clusters for the file. The starting cluster in the file entry in its containing directory and the file allocation table (FAT), must be corrected. Ideally one would have two copies of the pseudo-disk in memory - the original, and one for storing the corrected disk. The latter can be copied to the pseudo-disk file when it is complete. A less memory intensive approach might have to be used.

Compaction need only be performed on disks which have been updated. When a disk is opened for writing, the disk-ID must be logged in a file recording all pseudo-disks which have been

6 POSSIBLE EXTENSIONS TO THE MINIMAL FILE SERVER

The central function of the file server is the exchange of tracks and sectors between clients and the pseudo-disk files on the server's disk. As such, the minimal functions required of the file server have been identified as those involved with exchange of disk data, establishing user connections and the swapping of disks attached to drives.

However, other issues remain, such as the provision of printing services, the collection of student files for marking by the analysis machine and a condense-and-compact function for eliminating fragmentation within the pseudo-disks. In addition, for the server to be used in a more universal context outside the specialized education environment, it would be desirable that it can be extended to support a variety of functions associated with high level servers (cf Section 1.3).

This chapter attempts to establish how these functions could be implemented, and if they are feasible. In all of this it is important that the efficiency of data transfer is not effected through these functions overloading the capabilities of the server machine.

6.1 COMPACTION AND THE CONTIGUITY OF 'DISKS'

The implications for performance due to fragmentation within the pseudo-disks was discussed in section 2.3, and the need for a condense facility to remove this fragmentation was established. At the same time the pseudo-disk files can be compacted so that they contain only that part of the disk which contains data.

Compaction involves reorganizing sectors within the pseudo-disk, and altering file structure information (e.g. directory information), to reflect the new order. The compaction algorithm is therefore heavily dependent on knowledge of the file system of an operating system, so that this facility should only be provided for operating systems which would be widely used on the system.

In MS-DOS this involves finding each file on the pseudo-disk, and choosing new contiguous clusters for the file. The starting cluster in the file entry in its containing directory and the file allocation table (FAT), must be corrected. Ideally one would have two copies of the pseudo-disk in memory - the original, and one for storing the corrected disk. The latter can be copied to the pseudo-disk file when it is complete. A less memory intensive approach might have to be used.

Compaction need only be performed on disks which have been updated. When a disk is opened for writing, the disk-ID must be logged in a file recording all pseudo-disks which have been

altered. A compaction task would operate in the background, where a disk-ID is dequeued and the compaction algorithm performed if the disk is not in use.

6.2 ANALYSIS MACHINE LINK

Part of the envisaged system is a machine for the analysis of student programs. This would include the marking of test solutions, and provision for on-line analysis as an aid during laboratory sessions. The analysis machine has to use the file server to obtain the users' files and to return the results of the analysis to students. However, whereas the analysis process is performed on files, the link is not file-based, but orientated to the provision of pseudo-disks.

The analysis machine could obtain the users files by switching students' pseudo-disks into its drives from which it can read the files to be analysed, and on which it can record its results. Although this is the simplest approach to implement, some complications arise.

- The files to be analysed will need to have a *standard name* so that the analysis machine can locate them on the student's pseudo-disk.
- Pseudo-disks have the characteristic of allowing only single access in write mode. This is satisfied during the marking of tests, but when analysis is required as an on-line aid, both the student and the analysis machine require write access to the student's pseudo-disk. This means that the request for analysis should result in the user's disk being switched out of his drives and only switched back in once the analysis is complete. This leads to undesirable '*dead-time*' for the user.
- The analysis machine needs to read files from the pseudo-disk and thus will have to be running the same *operating system* as that under which the user's pseudo-disk is used. Since the analysis software runs under MSDOS [2], only MSDOS pseudo-disks could be analysed. Currently this presents no problem since all student assignments are associated with MSDOS, but it is undesirable to preclude future assignments based on other operating systems. Instead it would be preferable that a file - irrespective of the operating system in which it was produced - and not a pseudo-disk, be passed for analysis.

Alternatively, mechanisms must be provided which allow single files to be sent for analysis. The complication is how to achieve this given the block-orientated nature of the link.

One possible method would be to copy the required file to an improper pseudo-disk. The concept of an improper pseudo-disk was introduced to achieve a directory disk which could be accessed under any operating system (*cf* section 3.2.7). Such a disk does not contain files stored according to a particular operating system's file system, but contains a single file stored on consecutive sectors of the disk. Reading of and writing to the disk require direct sector accesses. When reading the file, there

is no way of determining how many sectors in the file, and where the file ends - which may be midway through a sector. Therefore, when the file is written, the first bytes should contain the byte-length of the file.

When a user issues a request for analysis, he specifies the file. A temporary pseudo-disk is created on the server - within an analysis section and named after the user and the file name - and is attached to one of the user's free drives. The required file is copied to the disk, which is then switched out of the drive, and the name logged into a list of files awaiting analysis. Once the analysis machine is ready, it attaches the disk to one of its drives and copies the file across to its own disk. The temporary pseudo-disk is then deleted.

Returning the results of the analysis, is a rough reflection of the above. Once complete, a new improper disk is created - with its name based on that of the original request - and the result file from the analysis copied to it. The user can at any time issue a request to recover the results of the analysis - the attempt will fail if the analysis is not yet complete. If successful, the result file is copied to the user's disk, and the result disk is deleted.

This removes the problems of being tied into a particular operating system, the need for standard file names and of user 'dead-time'.

6.3 PRINTING SERVICES

It is desirable to have a high speed printer as a shared resource on the network. This necessitates a printing service presented to each node, which schedules (and keeps isolated) individual printing requirements.

Once again however, printing, which is byte or file oriented, has to be achieved on block structured link.

It is attractive to trap the printer output at each node computer, and redirect this to the server, so that the network is transparent to existing software which uses the printer. However, it would be exceedingly inefficient for the network, if each byte sent to the printer was sent as a message to the server (since each messages entails writing a sector).

In addition, it is important that parts of printouts from different nodes are not interleaved. Instead, it is necessary that the entire print is accumulated before it is released for printing. This consideration does not exist in single user situations, where software sends data to the printer as it becomes available. The user will have to issue a command to perform the print of what has been accumulated.

The suggested approach is to have a single file improper pseudo-disk attached to a drive at the client, which is dedicated for printing purposes. Once again, the first part of the data can be used to indicate the length of the printout.

A routine at the client would have to trap and accumulate in memory, any data sent to the printer. As sufficient data is accumulated, sectors can be written to the current position on the print disk. When the user issues a command to complete printing, the remaining data in memory can be written to the print disk, and the correct length of the printout inserted at the start of the disk. A command is then sent to the file server, requesting a printout.

When the server receives the request, it must rename the file holding the pseudo-disk of the user's print disk and enqueue this file for printing. (A new pseudo-disk file must be created for the user's print disk).

Printing at the server involves dequeuing a file, interpreting its length, and sending it byte by byte to the printer.

6.4 FILE TRANSFER

As mentioned, the service is oriented towards the provision of pseudo-disks and not the transference of files. Yet certain file manipulation can sometimes not be achieved directly within this approach

A means of file transfer using a single file 'improper' pseudo-disk was introduced, for the collection of student's files to be analysed, and for printing purposes. This same technique, can be used to allow the *transfer of files between different operating systems and between different users*. The 'improper' disk is used as an intermediary. Since the transfer is not node-to-node, but by way of the file server, the transference of files between users can be well controlled from the server. This would allow transfers to be prevented during tests.

An *alternative* is to allow requests to the file server to refer to files on the pseudo-disk. Although the server views a pseudo-disk as a single file, it would be possible, given knowledge of the file structure on the pseudo-disk, to extract particular files. This feature could be provided for the most-used operating systems, where intra-pseudo-disk manipulation based on knowledge of the file structure is already necessary, to achieve compaction and contiguity (*cf* section 6.1).

This would allow the collection of files for analysis and the printing of files to be achieved somewhat more efficiently. Using this approach, a request would specify the file, which the server could extract from the pseudo-disk, and send *directly* for printing or analysis. The former method involved the user copying the file from his user disk to another, single file, pseudo-disk.

Effectively, this means that the file is transferred first to the user and then back to the file server, before the file was sent for printing or analysis.

6.5 SHARED DATABASES

Presently, concurrent use of disks is confined to when the users require read-access only. This places limitations in support of shared data-bases, where simultaneous access with the possibility of update, is necessary.

Support of shared data-bases would generally entail allowing for the concurrent access and update of these data-bases, by different users. Such concurrency control is given considerable support in high level file-servers [3, pp359, pp370]. Thus, while concurrent update was not in the original scope of the educational application, it is desirable that the system can be upgraded to support it should it become necessary - particularly if the system is to have more general application.

To allow the concurrent update of a pseudo-disk, it is necessary to ensure that a single consistent view of the disk is maintained on the network. This means that the distributed system of caching (i.e. caches at the disk emulators of client nodes) must be disabled for all disks supporting shared update. These caches allow the proliferation of multiple versions of the tracks of a pseudo-disk, since tracks held at an emulator would not be affected when another client updates those tracks at the file server.

Section 1.3.7 introduced the two-phase locking mechanism used to achieve synchronization of concurrent accesses. Here 'locking' obtains exclusive access to a region of data, while 'unlocking' releases the region. Software must adhere to the following characteristic for accessing regions.

- Request the required data region(s). If successful, the server locks the regions against further access.
- Complete the 'transaction' on the region.
- Release the region(s)

Once released, the data at the client must be considered void, and any further transactions involving it, must require another request to the file server.

This locking protocol can be used for concurrency control in the access of pseudo-disks. There are however some limitations in what can constitute a data region.

File-locking is unsuitable, because the link is disk-based, and the file server is ignorant of what constitutes a file at the client. This obviously precludes any regions which are portions of a file, such as database records within a file. Regions which can be controlled by the server, must therefore be physical blocks on the pseudo-disk - either the entire disk, a fixed block (track or sector), or a variable sized block.

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.