TR-059

The Concepts and Facilities of SIMPOS File System

by

Takashi Hattori and Toshio Yokoi

April, 1984

The Concepts and Facilities of SIMPOS File System

Takashi HATTORI and Toshio YOKOI

Institute for New Generation Computer Technology

Mita Kokusai Building 21F

1-4-28, Mita, Minato-ku, Tokyo 108, JAPAN

Abstract

This report describes the file system of SIMPOS, a programming and operating system for SIM (Sequential Inference Machine). The file system, one of the input/output medium systems, will provide permanent storages for data and objects in disk volumes.

Table of Contents

1. Introduction

SIMPOS is a programming and operating system for SIM (Sequential Inference Machine). It provides researchers with software development tools for logic programming. The operating system part of SIMPOS has three layers: the kernel, the supervisor, and the input/output medium systems.

The file system is one of the input/output medium systems. It manages disk storage. This report describes the concepts and facilities of the SIMPOS file system. The file system of SIMPOS has been designed to provide permanent storages for data, objects, and knowledge. By permanent storages we mean that even if the system goes down, data in the storages will remain intact and will be retrievable when the system is brought up next time.

(1) Data storage

The file system provides permanent storages for data, called files, as in a conventional file system. A file consists of records, which are the units of i/o operation of the file system. Data in main memory is written into a record in a file. Later the same data can be read from the record in the file into main memory.

(2) Object storage

The file system also provides object storages. Because SIMPOS is based on an object-oriented scheme, users may want to store objects they have created in disk storage so that they can be retrieved later. Examples of object storage use include the following:

o file

In a conventional file system, a file object (or a file descriptor) is stored in a VTOC (Volume Table Of Contents) which the file system deals with as a special file. In our scheme, the VTOC is treated as an object storage for file objects, that is, a special case of general object storages.

o user

Even in a personal computer, user information is necessary to set up the system in a manner suitable to each user. Such information is described by user objects which must be permanent. With an object storage for user objects, when a new user is catalogued, we simply create a new user object, initialize it with given information, and store it in the object storage. Later when this user logs in, the system retrieve this user object from the object storage.

o node

When a computer is connected in a computer network, information about each node is necessary. As with user objects, when a system connects to another node, it can retrieve a node object describing the remote node.

As these examples show, object storages are quite useful. Our file system provides a general mechanism for storing and restoring many classes of objects. A permanent storage for objects is called an instance file. A program can store and restore an object in an instance file.

A stored object may be identified by its name, as well as by its recording position. The directory file system is constructed so as to allow retrieval of a stored object with its name.

(3) Knowledge storage

The file system will also support the construction of data and knowledge bases. Since these are important in many application programs, especially within the FGCS project, a file system is expected to support some of the storage management features required for their implementation. Although we are well aware of this requirement, we will not fully incorporate it into the file system, because we have to keep the file system simple enough to be developed within the limitations of our time scale and man-power. Given these constraints, we feel it adequate for our purposes to concentrate on the storage and retrieval of data representing knowledge.

The method of storage depends on how the knowledge is represented. It may be represented as data, programs, or both. Objects give a good way to represent knowledge, because they can have both data and programs. For this purpose the file system has object storages.

With regard to efficient retrieval, databases usually have excellent capabilities, whereas file systems may not. A database system can only give a user necessary information by creating a new relation (a virtual file) from existing relations. To provide this facility, a database must be equipped with data dictionaries. It must also have a indexing mechanism in order to facilitate fast data retrieval. Our file system will provide a binder mechanism for this purpose. Using this mechanism, the user can construct virtual files from multiple files. It gives a kind of user view on files, as well as flexible record structures and accessing modes. The binder mechanism will not be described in this report, as its design has not yet been completed.

2. Volume Structure

Files are stored in a disk volume. In this section, the physical structure of a volume is described. In designing the volume structure, flexibility, was given priority over efficiency.

2.1 Volume

A volume (object) is defined to manage and access a physical disk volume. A disk device is represented by a disk (object), which accepts physical i/o operations and controls the disk device. The definition of disks is responsible to the device management of the kernel.

(1) Volume descriptor

Each disk volume has in its fixed area a volume descriptor that contains information describing the volume. It includes the volume identification, a VTOC (Volume Table of Contents), a root directory file, and a free page pool. Before any i/o operation is issued to a volume, this volume descriptor must be retrieved to initialize the volume (object). This procedure is called mounting a volume on a disk. Also when a volume is no longer needed, it may be dismounted. Class 'volume' has the following operations:

o To mount a volume on a disk

```
:mount(Volume,Disk)
```

o To dismount a volume

```
:dismount(Volume)
```

The logical structure of a volume is illustrated in Figure 2.1.

```
 _____________________________________
| volume descriptor                   |
|-------+---------------+-------------|
|       |               |             |
|   ____v______     ____v______       |
|  | VTOC      |   | root      |      |
|  | (file file)|  | directory |      |
|  |           |   | file      |      |
|  |           |   |           |      |
|  |___________|   |___________|      |
|                                     |
|   ___________                       |
|  | file      |                      |
|  |           |                      |
|  |___________|                      |
|                                     |
|_____________________________________|
```

Figure 2.1 Volume Structure

(2) Page allocation

Disk storage is divided into disk pages of fixed length. Some of these pages are used, and others are free to use. A volume allocates and de-allocates disk pages. Class 'volume' has predicates such as:

- o To allocate a free page

  :allocate(Volume,Page)

- o To de-allocate a page

  :deallocate(Volume,Page)

where 'Page' is a disk page address in the volume.

(3) Accessing

A volume defines read/write operations for the physical volume. A block of data is transfered between main memory and disk volumes. Only direct access to a disk volume is supported; a disk page address must be specified at each access. It is not assumed that volume access is used by user processes. Only the file manager (described later) uses it.

2.2 Region

A region is an area of a volume that stores records. It consists of disk pages allocated to it from the volume. A region object is defined to represent a physical region of a volume. Note that multi-volume regions (files) are not supported.

(1) Region descriptor

Each region in a volume is described by a region descriptor which includes information such as page table and region size. The region descriptor is stored in an entry of the VTOC, as part of the file descriptor. The file descriptor contains information such as the file type and record size, in addition to the region descriptor. Note that the VTOC is constructed as an instance file of file objects, and the first entry of the VTOC describes itself.

```
entry#      VTOC (file file)
        ---------------------------------
  0    | region des.   | file des.      |
       |---------------+----------------|
  1    |               |                |
       |---------------+----------------|
  :    |               |                |
```

Figure 2.2 VTOC (file file)

When a new region is created on a volume, a new entry describing this region is added to the VTOC. When an existing region is opened, its region descriptor is read to initialize a region object.

(2) Page table

Each region has a page table which maps the logical page address space relative to the region into a physical page address space relative to the volume. A page table itself occupies a page. An ordinary page table entry contains a physical page address associated with it. But the last few entries contain the addresses of the next level page tables, so that a region can be as large as the user pleases. Figure 2.3 shows the logical structure of a page table. The page size of a region is not fixed at creation, rather it can be increased dynamically by allocating new pages.

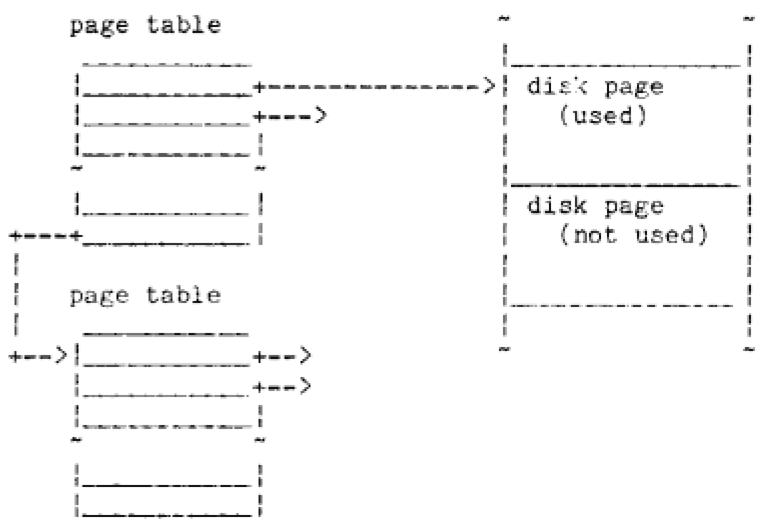

Figure 2.3 Page Table

(3) Accessing

A region (object) defines read and write operations for the physical region. A block of data in a region is directly accessed with a relative position in it. A given position is mapped to a physical position in the volume, and a read or write operation is issued on the volume with the mapped position.

3. Data Storage

A basic data storage is provided as a file. A file is viewed as an ordered collection of records; a record consists of many fields, each of which stores an item of data. The file system provides a means for storing and retrieving records in files, but it does not deal with fields in a record.

3.1 As a File

A file is an object which defines record access to a physical file stored in a region of a disk volume. The file system provides three basic types of files -- binary files, table files, and heap files -- which have different record formats. Each file type has two accessing modes; direct access mode and sequential access mode, both with positional and indexed accessing. The general features of these files are defined as a class named 'as_file', which is inherited by all types of file classes.

(1) Creating and opening a file

When a file object is instantiated, it is not yet associated with a physical file on a volume. An open operation associates a file with an existing physical file, and a create operation creates a new physical file and associates the file with it. When all accesses to an opened or created file have been completed, the file must be closed. When a physical file is no longer necessary, it can be removed to release its region pages. These operations defined on a file are:

- o To open a file

  :open(#file,File,File_Name)
- o To create a file

  :create(#file,File,File_Name)
- o To close a file

  :close(File)
- o To remove a file

  :remove(File)

'File_Name' is a string identifying a file on a volume. The specification of file names will be explained later.

(2) File buffer

A record in a physical file is read into a buffer in main memory, and a record data in a buffer is written into a physical file. A file buffer is an object containing an integer vector or a string (defined in KL0) for input/ output data. The reason why we define a buffer object rather than use a vector or a string directly, is that a buffer object hides how a record is constructed and gives the same interface for all file classes. A buffer is defined so as to have the following operations:

- o To get the record data in a vector/string

  :record_data(Buffer,Record_Data)
- o To get or set the data size

:data_size(Buffer,Data_Size)

:set_data_size(Buffer,Data_Size)

Other operations will be defined depending on the specific file record.

(3) File marker

A record is identified either by the record position or by a key associated with a record. The definition of a record position depends on the type of file, but in any case it corresponds directly to the location of the record in a file. File accessing using a record position will be called a positional access, and that using a key is called an index access. A file that allows an index access is classified as an index file.

A file marker is an object pointing to or identifying a record in a file. When accessing a file, a file marker is specified to indicate the record to be accessed. A file marker has two operations:

o To get the record position of the file marker

:postion(File_Marker,Position)

o To set the specified record position to the file marker

:set_position(File_Marker,Position)

Operations on a file involving a file marker include:

o To get the beginning_of_file file marker

:start_marker(File,File_Marker)

o To get the end_of_file file marker

:end_marker(File,File_Marker)

Note that the file marker is not updated even after the end_of_file has been changed.

(4) Direct access

Having defined a file buffer and a file marker, we can now introduce the operations for accessing a file. A file accepts direct-access read and write operations with a file marker such as:

- o To read a record at the file marker

  :read(File,Buffer,File_Marker)

- o To write a record at the file marker

  :write(File,Buffer,File_Marker)

Some other operations on a file include:

- o To add a record at the end of the file

  :add(File,Buffer)

- o To delete a record at the file marker

  :delete(File,Buffer,File_Marker)

In general, writing a record to a file requires the following sequence:

```
:record_data(Buffer,Record_Data),
    % Take out a record data vector or string to store the output
    % record data.
:set_data_size(Buffer,Data_size),
    % Specify the size of the output data.
:set_position(File_Marker,Position),
    % Set the output position.
:write(File,Buffer,File_Marker).
    % Finally issue a write operation on the file.
```

Reading a record requires the following sequence:

```
:set_position(File_Marker,Position),
        % Specify the input position.
    :read(File,Buffer,File_Marker),
        % Issue a read operation on the file.
    :record_data(Buffer,Record_Data),
        % Take out the record data.
    :data_size(Buffer,Data_Size).
        % If necessary, get the data size.
```

(5) Sequential access

Sequential access is provided by a file tap. A file tap is an object which maintains the current access position in a file and accepts the read and write operations at this current position. A file tap is created by requesting a file in the following operation:

o To create a tap on a file

```
:tap(File,Tap)
```

The tap is initially positioned at the beginning of the file. Internally, a file tap has a file object and a file marker. After a file tap has been created, sequential access is requested by operations on the tap, such as

o To read or write a record at the current position

```
:read(Tap,Buffer)
:write(Tap,Buffer)
```

Each time a read or write operation is performed, the current position of the file tap is advanced to the next record position. It is also possible to read or write a record without moving the file tap.

In addition to these read/write operations, a file tap provides several predicates to control the current position:

o To note the current position

```
:marker(Tap,File_Marker)

:beginning(Tap)

:end(Tap)
```

o To point to a new record position

```
:move(Tap,File_Marker)

:move_to_beginning(Tap)

:move_to_end(Tap)
```

A file tap is illustrated in Figure 3.1.

```
file tap             file (object)        file (physical)
___________          ___________          ___________
|___________+------>|           |======>|           |
|___________|       |           |       |           |
      |             |___________|       |           |
      v                                 |           |
file marker                             |           |
___________                             |___________|
|___________|==========================>|___________|
                                        |           |
                                        ~           ~
```

Figure 3.1 File Tap

3.2 Specific Files

In this section we explain several file types provided by the file system. Each is defined as a sub-class of a general file by inheriting class 'as_file'.

(1) Binary File

A binary file is a long string of fixed-length (usually one- or two-byte-long) records. Each record is identified by a record position within a file. If records are byte (two-bytes) long, the record position is a byte (two-byte) offset within the file. A binary file allows consecutive records to be read or written in a single access.

```
record
position    0         1         2         3
          ___________________________________...
          | data  |         |         |         |
          |_______|_________|_________|_________|_...
```

Figure 3.2 Binary File

A sequential access to a binary file is performed using a binary file tap. A binary file tap differs from an ordinary file tap in that it advances by as many record positions as are accessed, rather than only by one record position at a time.

(2) Table File

A table file is an ordered collection of fixed-length records. Since all the records in a table file are of the same size, the file itself can have a record size information. When a file is created, the record size must be specified. When an existing table file is opened, the data size of the file is given to the table file object. The physical record size is the same as the data size, that is, each record contains only user-provided data. Each record is identified by a record position, as usual.

```
           record
           position
                     ________________________
               0    |   data                 |
                    |------------------------|
               1    |                        |
                    |------------------------|
               :    |                        |
                    -                        -
```

Figure 3.3 Table File

A sequential access is performed using a table file tap.

(3) Heap File

A heap file is an ordered collection of variable-length records. A physical record consists of a type field which specifies the type of the record (string or vector), a length field which specifies the length of the record, and a data field. The general structure of a heap file is shown in Figure 3.2.

```
        record
        position
                  ______________________________
            0    | type  | length      | data     |
                 |-------+-------------+-----------
            1    |       |             |        |
                 |-------+-------------+-----------------
            2    |       |             |                 |
                 |-------+-------------+-----------------
            :    |       |             |
                 -       -             -
```

Figure 3.4 Heap File

Since the records in a heap file are of variable-length, it is not easy to locate a record simply by its position. When a position is given, the specified record must be searched for sequentially from the beginning of the file. Therefore, direct access to a heap file is not probable if it is specified by the record position alone. We define a heap file marker as having a byte offset, in addition to a record position. This byte offset is given in a file marker when a record is created (appended). Then, making use of this file marker, direct access to the record is really "direct".

A heap file tap is used to sequentially access a heap file. Note that moving a heap file tap to a desired position requires sequential search.

(4) Index File

An index file is a table file in which each record is associated with a key and it is specified by the key. Keys may be duplicated, that is, records having the same key can be stored in an index file. Since duplicated keys are allowed, a key is not enough to specify a record. The relative record number among the records having the same key is also necessary to identify a specific record.

A physical index record, which is a physical record in an index file, consists of a key field, a data field, and link pointer fields. The link pointer fields are used internally to chain records in a file. The size of these fields is fixed so that index records can be stored in a table file. The physical record size is the sum of these fields. The internal format of an index file is shown in Figure 3.3.

```
<------- record size ------->
 ___________________________
| links | key    | data        |
|-------+--------+-----------|
|       |        |             |
|-------+--------+-----------|
|       |        |             |
~       ~        ~             ~
```

Figure 3.5 Index File

An index buffer has additional predicates which deal with the key field:

o To get the record key

:record_key(Buffer,Key)

o To set the record key

:set_record_key(Buffer,Key)

An index file marker is an object that indicates a record in an index file. It contains the current key as well as the current record position. Operations on an index file marker include:

o To set a key

:key(File_Marker,Key)

o To get a key

:set_key(File_Marker,Key)

The behavior of a sequential access to an index file depends on whether we allow key ordering. If keys are ordered, records can be retrieved in either an increasing or decreasing order. For instance, to allow key ordering, we have to implement index files as a B-tree files. However, as it is fairly difficult to implement B-trees, the file system does not support key ordering. On the other hand, if keys are not ordered, only records having the same key can be sequentially retrieved. Such an index file can be fairly easily implemented with a hashing mechanism, and it will be provided in the file system.

4. Object Storage

This section describes the object storage facilities provided by the file system. We will explain how to store and restore objects in disk storage.

4.1 Instance File

An instance file is a file for storing objects. Each object is stored as an instance record in an instance file.

(1) Instance record

An instance record is a record containing sufficient information to restore an object. An instance file is a collection of these instance records.

If all the objects can be represented in fixed-length records, a table file can be used to implement an instance file. We call such an instance file a table instance file. On the other hand, if the instance records in an instance file must be of variable-length, a heap instance file is necessary to store them. Since there are many classes in the system, the instance records for these objects cannot be of the same size. Even the objects of the same class may be of different sizes. It is possible to store these objects in a heap instance file, but as it is not easy to manage such files efficiently, we have avoided their use. By placing some restrictions on objects to be stored, we can use instance table files instead. These restrictions are:

o Each class whose instances are to be stored in a file has an instance file.

o All the objects of that class must be represented in fixed-length instance records.

Operations for conversion between an instance record of fixed-length format and an object in main memory must be defined in each class. The file system, assuming these conversion operations are defined, calls them when storing or restoring an object. If they are not defined in a class, a store or restore operation causes an error.

An instance record is identified by a record pointer, just as an object is identified by an object pointer. A record pointer is itself an object containing a file where the instance record is included and a file marker where this record is positioned, so that an instance record can be uniquely identified in the system.

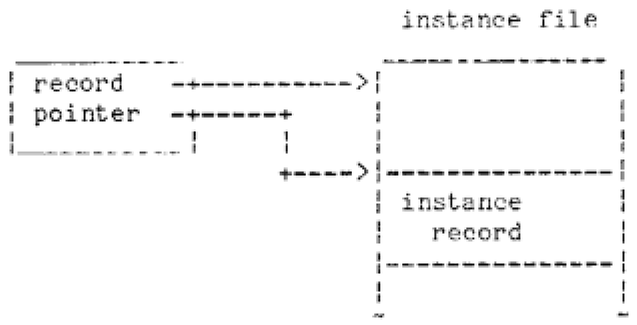

Figure 4.1 Instance Record

(2) Storing and restoring objects

A store predicate, which is an instance predicate, stores an object and returns the record pointer to the instance record of the object. A coding example is given below.

```
        :store(Object,Record_Pointer) :-
            :decode(Object,Instance_Record),
                % Convert Object into Instance_Record.
            :instance_file(Object,Instance_File),
                % Get the instance file of the class of Object.
            :add(Instance_File,Instance_Record,File_Marker),
                % Write out Instance Record. The position of the written
                % record is returned in File_Marker.
            :create(#record_pointer,Record_Pointer,Instance_File,File_Marker).
                % Create a record pointer identifying the instance record.
```

A restore operation is defined as a class predicate. It takes a record pointer and returns a restored object. A coding example is given below.

```
        :restore(Class,Object,Record_Pointer) :-
            :file(Record_Pointer,Instance_File),
                % Get Instance_File.
            :file_marker(Record_Pointer,File_Marker),
                % Get File_Marker.
            :read(Instance_File,Instance_Record,File_Marker),
                % Read in Instance_Record.
            :object_class(Instance_File,Class),
                % Get Class of the objects in Instance_File.
            :new(Class,Object),
                % Create a new instance.
            :encode(Object,Instance_Record).
                % Convert Instance_Record to Object.
```

These two predicates are defined in class 'as_permanent_object'. A new class whose instances are permanent can be defined by inheriting this class and defining its own encode/decode predicates.

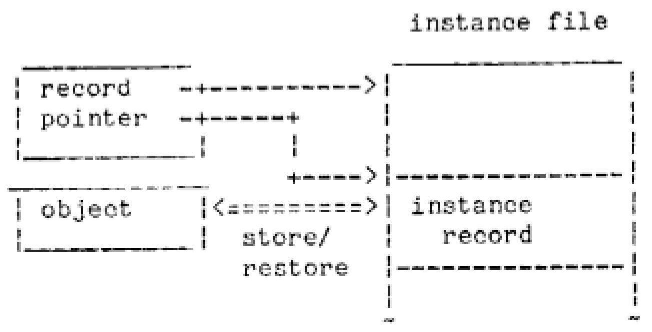

Figure 4.2 Storing and Restoring Objects

4.2 Directory File

An instance record representing an object in a file is identified by a record pointer, as mentioned above. However, it is not probable that a user would remember this record pointer. Therefore, an instance record (an object) will be able to be identified by its name instead. A directory file system provides a means for storing and retrieving objects in files using their names (represented as character strings).

(1) Directory file

A directory file is an index file that associates the names of objects with record pointers. After an object is stored with its name, it can be retrieved with the name, instead of with its record pointer. Class 'directory_file' defines several operations, such as:

- o To bind a record pointer with a name

  ```
  :bind(Directory_File,Record_Pointer,Name)
  ```

- o To find a record pointer with a given name

  ```
  :find(Directory_File,Record_Pointer,Name)
  ```

- o To replace an old record pointer with a new one

  ```
  :replace(Directory_File,Record_Pointer,Name)
  ```

- o To rename a name associated with a record pointer

  ```
  :rename(Directory_File,Name,New_Name)
  ```

For example, to retrieve an object with a given name:

```
:find(Directory_File,Record_Pointer,Name),
    % Find Record_Pointer with Name.
:restore(#as_permanent_object,Object,Record_Pointer).
    % Restore Object from Record_Pointer.
```

The directory file is illustrated in Figure 4.3.

```
            directory file
   ______________________________
   | name  | record pointer     |
   |-------+--------------------|
   |       |                    |
   |-------+--------------------|
   |       |                    |
   ~       ~                    ~
```

Figure 4.3 Directory File

(2) Directory file tree

As any class of objects can be bound in a directory file, another directory file may be bound. A directory file tree is, therefore, constructed. To identify an object (instance record) from the root of this tree, a pathname is used instead of a simple name. A pathname is of the form, for example,

"dir1>dir2>name"

where "dir1" is the name of a sub-directory file of the root, "dir2" is the name of a sub-directory of the directory file "dir1", and "name" is the name of the object specified. The object (instance record) specified by this pathname is illustrated in Figure 4.4.

```
directory file (current)
 ___________
|       |   |   directory file (dir1)
|-------+---|    __________
| dir1  |   +--->|      |   |   directory file (dir2)
|-------+---|    |------+---|    __________     instance file
|       |   |    | dir2 |   +--->|      |   |   __________
-       -   -    |------+---|    |------+---|  |          |
                 |      |   |    | name |   +--->|----------|
                 -      -   -    |------+---|  | inst.rec.|
                                 |      |   |  |----------|
                                 -      -   -  |          |
                                               -          -
```

Figure 4.4 Directory File Tree

A directory file knows the pathname convention and provides the following operations:

o To retrieve a record pointer with a pathname

    :find(Directory_File,Record_Pointer,Pathname)

o To bind a record pointer with a pathname

    :bind(Directory_File,Record_Pointer,Pathname)

(3) Volume root directory file

Each volume has a root directory file. All the directory files on the same volume are under this root directory file. When a volume is mounted, its root directory file is given. Using this root directory file, we can retrieve any object in the volume with their pathnames.

4.3 Directory System

In SIMPOS, the directory file system is part of a general directory system, which is provided by the supervisor. This means that objects in files can be specified with with pathnames just as is done for objects in main memory. This section gives a brief description of how the file system is incorporated into the directory system.

(1) Directory

A directory is an index pool that associates objects with names in main memory. A directory tree can be formed by placing sub-directories under a directory. A pathname is used to specify an object in a directory tree.

The directory system maintains a system root directory and a process root directory for each process. A pathname has two formats:

- o To specify an object from the system root directory

  e.g. ">dir1>dir2>name"

- o To specify an object from the process root directory

  e.g. "dir1>dir2>name"

A global object has the following operations:

- o To bind an object with a pathname

  :bind(Object,Pathname)

- o To retrieve an object with a pathname

  :retrieve(#as_global_object,Object,Pathname)

where class 'as_global_object' is defined to provide an object with the pathname convention. As we see here, a directory object does not appear explicitly; an object is identified only with its pathname.

(2) Linking a volume

When a volume is mounted, its root directory file is restored. This volume root directory file can be inserted into the directory system by a predicate:

- To link a volume into a directory system

```
:link(Volume,Pathname)
```

For example, the operation

```
:link(Volume,">disk1>volume1")
```

associates the root directory file of 'Volume' with the pathname ">disk1>volume1". Then, a file with the name "dir1>filename1" on this volume is retrieved by

```
:retrieve(Object,">disk1>volume1>dir1>filename1")
```

just as a global object is.

(3) Permanent directory

When a directory file tree is inserted as a sub-tree of the directory tree, permanent objects can be specified with their pathnames in the same way as global objects are. However, this approach has two problems:

- File access is required every time an object in a directory file is retrieved.
- The same object may be retrieved more than once through a directory file. This jeopardizes object consistency.

A permanent directory is defined to solve these problems. A permanent directory has a directory file as its permanent storage. When being retrieved through a permanent directory for the first time, an object is retrieved from the accompanying directory file, and, at the same time, it is itself inserted into a directory. Then the same object is retrieved from the directory, not from the directory file. This technique circumvents unnecessary file access, and also it prevents object duplication in the system.

```
      permanent directory ----+
   ________________________   |   directory file
  | name  | obj.pointer    |  v________________________
  |-------+----------------|  | name  | rec.pointer    |
  |       |       .        |  |-------+----------------|
  |-------+-------+--------|  |       |       .        |
  |       |       |        |  |-------+-------+--------|
  -       -       |        -  |       |       |        |
                  |           -       -       |        -
                  |                           |
           _______v_______     restore        |
          | perm.object   |<==========  ______v_______
          |_______________+----------->| inst.record  |
                                       |______________|
```

Figure 4.5 Permanent Directory

Generally, each directory file will be associated with a permanent directory. When a directory file owned by a permanent directory has a sub-directory, retrieving the sub-directory file through the permanent directory creates a permanent sub-directory.

5. File Manager

In this section, we explain how the file system is implemented. The file system is under the control of a process, called a file manager, whose main task is to dispatch i/o requests from the customers to the device handlers, and i/o replies from the device handlers to the requesting customers. The file manager has two interfaces, the customer interface and the device interface.

5.1 Customer Interface

A customer is a process which requests i/o operations on files. From the customer's view point, all the i/o operations are performed on a file object. However, internally, an operation sends a request message to the file manager and receives a reply message from it when the i/o operation has been completed.

(1) Manager channel

The file manager has a channel, called a manager channel, for receiving messages from customers. This channel is created and kept in a class slot when the file manager is instantiated.

(2) I/O port

A file inherits class 'io_port', which is a special type of port. When a file is instantiated, a file is connected to the manager channel of the file manager. When a customer performs an operation on a file object, the file object assembles a message and sends it to the file manager.

```
customer                         file manager
    :                                 :
    O     _       (request)       _   O
   -+-   | |-------------------->|_| -+-
    |    | |      (reply)         :   |
   / \   |_|<--------------------- / \
          :                       :
         io port             manager channel
         (file)
```

Figure 5.1 Customer Interface

5.2 Device Interface

The device interface is an interface between the file manager and a device (disk) handler.

(1) Interrupt process

A physical device is controled by a device handler, which is an interrupt process. An interrupt process is dispatched differently from ordinary processes, as follows:

- o When an interrupt takes place, an interrupt process associated with it is dispatched by hardware.
- o When a message is sent to a channel that an interrupt process is waiting for, the channel raises a trap to activate the interrupt process.
- o When the interrupt process finishes the requested operation, it releases the processor so as to re-dispatch the other processes.

A device handler is instantiated and activated during a system boot-up procedure.

(2) Handler channel

Each device handler has a channel, called a handler channel, to receive messages from the file manager. The file manager communicates with the device handlers by sending messages through this channel.

(3) Device port

A volume (object) is a device port connected to a handler channel of the device handler. An i/o operation on a volume is converted to a message to the device handler.

```
file manager                       device handler
    :                                   :
    O    _       (request)          _   O
   -+-  | |--------------------->|_|  -+-
    |   | |      (reply)            :    |
   / \  |_|<----------------------- / \
         :                          :
    device port              handler channel
    (volume)
```

Figure 5.2 Device Interface

5.3 Process Structure

The file manager is a process that executes the following procedure.

- o It receives a message either from the manager channel or from one of the volumes (device ports).
- o If the received message is from the manager channel, it is an i/o request from a customer. The manager interprets this message and requests a physical i/o operation on the volume. The volume then sends a message to the handler channel.
- o If the message is from a volume, it indicates the completion of the requested i/o operation. The file manager posts it to the requesting customer.

The interactions among the file manager, customers, and device handlers are illustrated in Figure 5.3. Though not shown in this figure, many customers can exist, each of which can have an unlimited number of files, and device handlers exist as many as physical disk devices.

```
customer                      file manager                  device handler
    :                               :                              :
    O    _      (request)      _    O    _      (request)      _   O
   -+-  | |------------------>|_|  -+-  | |------------------>|_| -+-
    |   | |     (reply)        :    |   | |     (reply)        :   |
   / \  |_|<-------------------    / \  |_|<-------------------   / \
         :                     :         :                     :
        io port              manager  device                handler
        (file)               channel  port                  channel
                                      (volume)
```

Figure 5.3 File Manager

A file manager is instantiated (created) and activated during a system boot-up procedure.

6. Further Discussion

We have described the concepts and facilities of the SIMPOS file system. Based on these concepts, we have completed the functional specification and implementation as class definitions. Our adoption of the class mechanism as a programming and specification tool enabled us a flexible and extendable construction of the file system for a fairly short time (one year). The total size of the coded ESP program is estimated to be 3K lines. We are now cross-debugging the file system on the development system.

Facilities which are not included in this report include:

- o Floppy disk files

  Floppy files are defined as to have only limited part of the facilities of files explained above.

- o System boot-up disk

  A system disk has a boot-up region that stores CSP (Console Processor of PSI, a personal version of SIM) programs, PSI firmware programs, IPL (Initial Program Loader), and so on. CSP reads in these programs during system boot-up. The file system supports access to this region, so that these programs can be updated within SIMPOS.

Works remains to be done on design and implementation in the following areas:

o File class coercion

A created file of a certain class is always opened as of the same class. However, it is sometimes useful if we can open a file as of another class. File class coercion will provide this facility.

o File manipulator

A file manipulator is an integrated utility to manipulate files using a window.

o Binder mechanism

A binder constructs a virtual file from ordinary files.

This work will be started when our current implementation has been completed.

Acknowledgements

We would like to thank all the members of ICOT 3rd. Lab for their contribution to the design of the file system, and the implementation group for the detail design.

References

[1] T.Chikayama, "KL0 Reference Manual", to appear as ICOT TR.

[2] T.Chikayama, "ESP Reference Manual", ICOT TR-44, Feb. 1984.

[3] T.Hattori, et al., "SIMPOS: An Operating System for a Personal Prolog Machine PSI", to appear as ICOT TR.

[4] T.Hattori and T.Yokoi, "The Concepts and Facilities of SIMPOS Supervisor", to appear as ICOT TR.