The File System of a Logic Operating System

Anthony J. Kusalik

Technical Report 84-21

November 1984

The File System of a Logic Operating System

Anthony J. Kusalik

Computer Science Department
University of British Columbia
Vancouver, B.C., Canada V6T 1W5

Technical Report 84-21
November 1984

ABSTRACT

This paper describes the file system of an operating system for a logic inference machine. The file system is composed of a file system device and a collection of file system servers. The former provides the basic services of creation, access (reading or writing), removal, and stable storage of files. It realizes a simple, though powerful model: a file store as a special type of name server maintaining associations between identifiers and entities. A file is then a pair, <file name, file content>, of terms. Clients gain access to a file by sharing the file content term with the file system device. Reading the file corresponds to examination of the term; writing, to instantiation. There is no need of explicit read or write operations, or of file closure. File system servers enhance or modify this basic file abstraction. They can provide features of more conventional file systems, such as hierarchical directories or fixed, structured file formats.

Concurrent Prolog is assumed as the underlying machine language and the operating system implementation language. However, the ideas are also applicable to other parallel logic programming languages, such as PARLOG.

As a prerequisite to describing the file system, the Concurrent Prolog machine model is presented, as well as an overview of the entire operating system design.

1. Introduction

The evolution of "Fifth Generation" computers requires novel approaches to computer languages, architecture, and applications [Moto-oka 82]. No less important than other areas, efforts in systems programming and operating systems call for innovation and pragmatism.

Recent work by the author has concentrated on the design of a logic operating system, an operating system intended for a logic inference machine and implemented in a logic programming language. Concurrent Prolog [Shapiro 83a], hereafter denoted "CP", is the implementation language chosen, and an abstract CP machine serves as the target hardware. Though the design is not yet complete, many concepts and features have been determined. Certain portions of the logic operating system have been implemented and tested using a CP interpreter.

In this paper the file system is outlined. The operation of a computer system is often critically tied to the methods used to store, retrieve, and manage information. A file system is therefore of great significance in characterizing an operating system: it is generally indicative of the overall design philosophy; it usually serves as a basis for other software; and in describing it, many other aspects of the operating system are revealed.

The applicability of the concepts in this paper are not restricted by the use of CP. Other logic programming languages, such as PARLOG [Clark & Gregory 84] or Flat Concurrent Prolog [Shapiro 84], would serve equally well. No special property of CP is utilized within the machine or file system models or the program segments describing the file system components.

This presentation is structured as follows. The **Introduction** provides a brief review of related work, a description of the assumed CP machine model, and an overview of the operating system. The bulk of the work, the description of the file system, is given in Section 2. The feasibility of implementation is discussed in Section 3, together with motivation for several facets of the design. Alternatives to various aspects are the subject of Section 4. Section 5 summarizes and concludes the paper.

1.1. Related Work

Though much published research in associated areas (logic programming languages, machine architectures, etc.) has appeared, only a small amount concentrates on systems programming and operating systems for inference machines. Shapiro [1983c] presents a feasibility study of CP as an operating system kernel language. A number of common, representative operating system functions are specified in the language. However, no attempt is made to describe a single, complete operating system. The operation of a peripheral device is described as a process, with the device "contents" regarded as an argument in the process state. It is proposed that the cleanest way to implement I/O functions in a CP machine is for peripheral devices to consume or generate CP streams. Clark and Gregory [1984] demonstrate that PARLOG, a language related to CP, is also an attractive systems programming language.

Hattori and Yokoi [1983] present basic concepts and constructs of SIMPOS (Sequential Inference Machine Programming and Operating System) for PSI (Personal Sequential Inference Machine). Further details are provided by Takagi et al. [1984], including a very brief, general description of the file system. Unfortunately, the design of PSI is not suited to the proliferation of small-sized processes characteristic of CP [Yokota et al. 83]. The machine language, KL0, and the implementation language, ESP, are forms of Prolog with depth-first search, backtracking, and "cut" [Chikayama 83].

Hence, some fundamental aspects of SIMPOS are incompatible with a CP environment.

1.2. Concurrent Prolog

CP was chosen as the implementation and machine language for the logic operating system because it is powerful, concise, and supports concurrent computation. Many effective programming constructs and techniques, such as objects, class hierarchies, stream communications, and message-passing, can be cleanly realized using the language [Shapiro 83a, Shapiro 83c, Shapiro & Takeuchi 83, Takeuchi & Furukawa 83].

It is assumed that the reader is familiar with CP. An in-depth description is provided by Shapiro [1983a]. Papers by Shapiro [1983c], and Shapiro and Takeuchi [1983] provide summaries. A computational model for a CP machine is given by Shapiro [1983a]. The same author also presents ideas on the basic architecture of such a machine [Shapiro 83b].

1.3. Concurrent Prolog Machine

The characteristics of the target CP machine are very important to the operating system and file system designs. As such a machine does not yet exist, it is necessary to define and assume a hardware model. The model is summarized as follows.

A CP machine consists of an arbitrary number of individual processing elements. Each process of a conjunctive goal system can be thought of as executing on an individual processor. The machine is responsible for mapping processes to available processors†. Hardware supports the efficient access and propagation of shared variable bindings.

Each physical I/O device has associated with it a special "device processor". This processor provides an interface between the remainder of the CP machine and the device. Viewed by other processing elements, a device processor supports a single, characteristic, perpetual process called a "device process" (DP). A DP is logically indistinguishable from other CP processes, and describes the operation of a device processor and I/O device. Software access to an I/O device (through its device processor) is achieved by communicating with the corresponding DP using CP streams. Device processes may vary in specific protocol details.

The machine language of the CP machine may be CP, in which case its operation is described by a meta-interpreter. Alternatively, the hardware may execute a logic-based language in which higher-level logic-based systems programming languages can be specified (cf. ESP and KL0 [Chikayama 83]). A CP interpreter is then written in this low-level language, but viewed as part of the machine. In either case, unification and goal reduction are provided by the machine model.

1.4. Operating System Overview

The operating system design follows principles of multi-process structuring (program structuring using multiple concurrent processes). Several characteristics of CP make this approach attractive: large numbers of small processes, easily attained interprocess communication, dynamic process creation and destruction, and the ability to share data structures among processes. The operating system resembles Verex [Cheriton 79, Lockhart 79], a multi-process structured system for a conventional architecture.

---

† Shapiro [1983b] presents an alternate view in which CP programs are augmented with process-to-processor mapping notations. The operating system design would not be adversely affected by such a change.

The logic operating system is composed of small, complementary, and cooperating servers. Each server provides a compact set of related services to other processes. Servers are generally constructed as CP objects, or occasionally as object hierarchies. They may dynamically create and destroy constituent processes. Servers communicate via object-based protocols using message-passing over streams. They may each call on the utilities of devices and other servers in their operation. Progressively more substantive services are generated in this manner. Clients normally communicate directly with the server responsible for the utility being sought. Servers may be transient (dynamically created to fill a temporary need, then removed) or permanent (created at system initialization for the duration of system execution).

The operating system does not include a kernel. Fundamental capabilities such as unification (for data transfer and message-passing), process creation, and process destruction are described by metalanguage and incorporated into the CP machine model.

In a conventional operating system, the bulk of the software cannot access physical I/O hardware directly. A device driver is introduced to provide an interface. Here, directly accessible devices are provided by the machine model. Clients may access an I/O device by communicating with its corresponding device process. The operating system need only assist in identifying the appropriate stream.

A device process exists independently of the operating system servers; it exists whenever its device processor is active. During system initiation (bootstrapping), the operating system obtains a channel to each device process. These channels are preserved for the duration of system execution.

2. File System

The file system of the logic operating system is composed of a file system device (FSD) and a set of servers. The FSD is the lowest-level component and is discussed here at length. Individual file servers are not described in detail; they will be the subject of future papers. However, examples of the types of supplementary service that could be provided are given, as well as program segments illustrating how they might be realized.

2.1. File System Device

The fundamental purpose of any file system is providing user applications and other system software with the ability to store and retrieve information in identifiable, nonvolatile entities called files. The design of the file system device (FSD) addresses directly this fundamental capability. The result is a simple, though powerful, file abstraction.

The FSD is a device process having the characteristics of a name server†. It is supported by a device processor and maintains a nonvolatile database of <file name, file content> associations as part of its state. The associated physical device is some form of stable storage. Individual associations (files) are affirmed, accessed, or recalled on receipt of appropriate messages.

In its fundamental form, the FSD is specified as

---

† A name server is a common operating system component which sustains a set of "entity" and "identifier" associations, and facilitates the modification and query of them.

```
fsd( [Req|Instrm], FileSysDB ) :-
  process_request( Req, FileSysDB, NewFileSysDB ),
  fsd( Instrm?, NewFileSysDB? ).
```

where *process_request* represents the clauses necessary for each file operation. The first argument of *fsd* is a shared communication variable. The second is the file system data structure. A file is a pair, *(FName,FContent)*, of terms recorded in this data structure. *FName* must be ground and serves as the file identifier. Using simple atoms as identifiers provides a flat name space. Structured terms result in a hierarchical naming scheme. *FContent* is the file content and is an arbitrary term. Communications from clients are in the form of incomplete messages.

The FSD offers a nontrivial, easy-to-use set of services. To create a new file, the request

affirm(FName, FContent)

is sent to the FSD. Its operational meaning is "affirm the association between the identifier *FName* and the term *FContent*". In response, the device process adds *(FName,FContent)* to its state. There is no need for messages or special operations to write into the file: the client simply instantiates *FContent*. Since the variable is shared by the FSD, any instantiations made to it (writes) are propagated to the device process. Unification is responsible for the actual data transfer. The active participation of the FSD is not required.

For an existing file, a different request is used:

access(FName, FContent)

Again *FName* identifies the desired file. *FContent* is instantiated by the FSD to the current content of the file. Reading is then the examination of the term bound to *FContent*. Read requests of the device process are unnecessary. The file is modified by instantiating variables in the file content term.

One last type of message is needed:

recall(FName, FContent)

It requests that the device process query its database for a file identified by *FName*. The file content is unified with the second argument, *FContent*. The association of *FName* and *FContent* is subsequently removed from the FSD's internal state, as suggested by the predicate name. A typical use of such a request is removing a file:

recall(FName, _)

A minimal FSD with this functionality can be concisely specified:

```
fsd( [affirm( FName, FContent )|ReqStrm], FileSysDB ) :-                    (a1)
    add_file( FName, FContent, FileSysDB, NewFileSysDB ),
    fsd( ReqStrm?, NewFileSysDB? ).
fsd( [access( FName, FContent )|ReqStrm], FileSysDB ) :-                    (a2)
    find_file( FName, FileSysDB, FContent, _ ),
    fsd( ReqStrm?, FileSysDB ).
fsd( [recall( FName, FContent )|ReqStrm], FileSysDB ) :-                    (a3)
    find_file( FName, FileSysDB, FContent, NewFileSysDB ),
    fsd( ReqStrm?, NewFileSysDB? ).

find_file( FName, [(FName,FContent)|FileSysDB], FContent, FileSysDB ).      (a4)
find_file( FName, [F|FileSysDB], FContent, [F|NewFileSysDB] ) :-             (a5)
    otherwise |
    find_file( FName, FileSysDB?, FContent, NewFileSysDB ).

add_file( FName, FContent, FileSysDB, [(FName,FContent)|FileSysDB] ).       (a6)
```

Program (a): File System Device

For clarity, the file system data structure is portrayed as a simple list. Use of a more practical structure, such as a tree, requires modification of the local predicates *find_file* and *add_file* only. The basic form of the program, clauses (a1), (a2), and (a3), remains unchanged. The program is easily translated to other parallel logic programming languages such as PARLOG (cf. the file store manager program of Clark and Gregory [1984]).

The program semantics are straightforward. The ***fsd*** process is executed with two arguments, a communication stream represented by *ReqStrm*, and an internal database, *FileSysDB*. It is normally suspended, awaiting a message. If an *affirm* request is received, the pair *(FName,FContent)* is added to *FileSysDB* to form *NewFileSysDB*. On a *recall* request, a *find_file* process is invoked with *FName* and *FileSysDB*. It returns the appropriate file content bound to *FContent* and the database, less the *(FName,FContent)* pair, bound to *NewFileSysDB*. In both cases the ***fsd*** process recurses with the remainder of the input stream and *NewFileSysDB*. On an *access* request, ***fsd*** again uses *find_file*, but retains the existing file data structure. Finally,

find_file(FName, FileSysDB, FContent, NewFileSysDB)

names the relation "*NewFileSysDB* is *FileSysDB* with the item *(FName,FContent)* removed".

With this view of file storage, a file is logically "open" as long as its file content term is being shared by a client and the FSD. A file is "closed" when no part of the term is being shared. The device process is oblivious to a file being open or closed.

Any number of processes may access a file simultaneously. Multiple access is achieved by an existing client or the FSD sharing the file content term with other processes. If there are multiple attempts to write, all "writers" must agree on the binding(s) made, as with any case of concurrent processes instantiating the same shared variable [Shapiro 83a]. Otherwise unification fails for all the writers.

The ability to write a file is not enforced by the file system device. Rather, a file may be written only if some portion of it is uninstantiated, i.e. if one of its constituent terms is a variable. For example, if the file content is the list of atoms

[t,h,i,s, ,f,i,l,e, ,h,a,s, ,s,e,t, ,c,o,n,t,e,n,t]

alteration is not possible. If it is instead

[a,p,p,e,n,d, ,t,o, ,t,h,i,s, ,f,i,l,e|X]

further instantiations can be made.

An obvious shortcoming of program (a) is its susceptibility to failure. For example, the *fsd* process fails if *FName* does not unify with an identifier in *FileSysDB*, or if an unrecognized request is received. Also, there is no check made for duplicate or uninstantiated identifiers. These potential problems can be solved using previously described techniques [Shapiro 83a]. For example, to prevent failure on an unrecognized message, the following clause can be added:

fsd([Req|ReqStrm], FileSysDB) :-
otherwise | fsd(ReqStrm?, FileSysDB).

A practical enhancement of program (a), especially in the case of error, is to include an explicit reply in each type of request. This requires modest changes to *fsd*, including a change in message format. Known techniques [Shapiro 83a] are again sufficient. For example, the form of an *access* request can be altered to

access(FName, FContent, Reply)

where *Reply* is bound by the FSD to either *success* or *failure*. The corresponding modification to *fsd* is:

fsd([access(FName, FContent, success)|ReqStrm], FileSysDB) :- (a2′)
find_file(FName, FileSysDB, FContent, _) |
fsd(ReqStrm?, FileSysDB).

fsd([access(FName, FContent, failure)|ReqStrm], FileSysDB) :- (a2″)
otherwise |
fsd(ReqStrm?, FileSysDB).

It may appear that the *fsd* process is characteristic of a CP object, with *FileSysDB* corresponding to the internal state. This is not strictly correct. Certainly, insertion and deletion of *(FName,FContent)* pairs is performed only by the process on receipt of appropriate messages. But *FContent* terms are meant to be shared with client processes. This violates the requirement that an object's internal state be operated upon from the outside only by sending a message to the object. Despite this, it is sometimes convenient and illustrative to treat *fsd* as an object.

The presence of more than one FSD in a computer system poses no difficulties. Device processes, as any CP processes, are distinguishable by their communication channels. Clients specify a particular FSD by the channel on which they select to send a message.

2.2. File System Servers

The FSD supports the basic services of file creation, access (reading or writing), removal, and stable storage. Servers extend this functionality. As a result, the logic operating system may provide a variety of alternate file abstractions, including diverse file formats, naming schemes, access methods, etc. The nature of the file system as perceived by a client process depends upon which file system entity it is communicating with. Users may introduce their own servers which add localized, custom features.

This approach allows a great deal of flexibility. A multitude of differing servers and file abstractions are possible. This section describes some, especially those of a more conventional nature.

Structured terms can be used as file identifiers to realize a hierarchical file naming scheme (UNIX pathnames [Ritchie & Thompson 74] are one example). To alleviate the need to always specify complete hierarchical names, users can employ a server. Such a server could be designed as follows. It retains a "base name" as its internal state. It combines this with the name specified in each FSD request channeled through it. The base name can also be replaced. The server might take the following form:

```
file_server( [change_base_name( BaseName )|ReqStrm], OldBaseName, FsdStrm ) :-
    file_server( ReqStrm?, BaseName, FsdStrm ).

file_server( [access( ShortName, FContent )|ReqStrm], BaseName, FsdStrm ) :-
    construct_full_name( BaseName, ShortName, FullName ),
    send( access( FullName?, FContent ), FsdStrm, NewFsdStrm ),
    file_server( ReqStrm?, BaseName, NewFsdStrm ).
```

Program (b): Server to aid with hierarchical names

The server process has three arguments: a channel for incoming messages, the base name retained as its internal state, and a channel to the file system device. On a *change_base_name* request, it replaces the base name with that specified in the message. For other requests – only the *access* case is shown above – the server constructs a full hierarchical name from the base name and the name in the message. It then forwards the modified request to the FSD. The predicate

send(Msg, Strm, NewStrm)

names the relation "the result of sending *Msg* on stream *Strm* is stream *NewStrm*" and is specified by

send(Msg, [Msg|Strm], Strm).

[Shapiro 83c].

The FSD allows file contents to be arbitrary terms, such as atoms, variables, lists, difference lists, trees, and other, more complex constructions. A file system server can use these data structures to provide more conventional file formats, such as character stream files, indexed or keyed files, and record format files. Different formats can be supported simultaneously by different servers.

Since variables are single assignment in logic programming, it is not possible to update nonvariable portions of files stored by the FSD. To update, the existing file must be discarded and a new version created†. However, a server can emulate a more conventional view of the file system, where updates are possible, and separate requests are necessary for opening, reading, writing, and closing a file. On an *open* message, the server recalls the desired file from the FSD and retains it as part of its internal state. *read* and *write* requests are processed with respect to this local information. On a *close*, the server affirms the revised form of the file.

---

† This characteristic is compatible with new laser disk technology in which data can only be written once. Also, it is consistent with the behavior of many conventional operating system utility programs which read a file, modify the content, and rewrite the file in its entirity (e.g. *ed* editor of UNIX).

The following program is a simplified example of a file server with this functionality:

```
file_server( [open( Name )|ReqStrm], _, _, _, FsdStrm ) :-
    send( recall( Name, File ), FsdStrm, NewFsdStrm ),
    file_server( ReqStrm?, Name, File?, [], NewFsdStrm ).

file_server( [read( Atom )|ReqStrm], Name, [Atom|RestOfFile], RevFile,
                                                        FsdStrm ) :-
    file_server( ReqStrm?, Name, RestOfFile?, [Atom|RevFile], FsdStrm ).

file_server( [write( Atom )|ReqStrm], Name, [OldAtom|RestOfFile], RevFile,
                                                        FsdStrm ) :-
    file_server( ReqStrm?, Name, RestOfFile?, [Atom|RevFile], FsdStrm ).

file_server( [close|ReqStrm], Name, RestOfFile, RevFile, FsdStrm ) :-
    append( RestOfFile, RevFile, FinalRevFile ),
    reverse( FinalRevFile?, NewFile ),
    send( affirm( Name, NewFile? ), FsdStrm, NewFsdStrm ),
    file_server( ReqStrm?, _, [], [], NewFsdStrm ).
```

Program (c): Server providing four customary file operations

The file abstraction maintained by the server is of a file as a sequence of atoms. Read/write operations are sequential and operate on single atoms.

The semantics of the program are straightforward. The first argument of *file_server* is a stream of incoming requests; the second, the file name; the next, the previous contents of the file; the fourth, the revised contents; and the final argument, a communication stream to the FSD. The server is initiated by an *open* request. It processes *read* and *write* messages until receiving a final *close*. A *read* request causes the next atom in the file to be returned to the client. The atom is also added to the revised file content. On a *write* request, the next atom is discarded, and the one supplied by the client is added to the revised file content. On receiving a *close*, the server appends the remaining contents of the file to the revised content, reverses it to form the new file content, and stores the new file. This last append operation would be more efficient, and the reversal unnecessary, if a difference list was used.

Binding large data structures to variables and propagating such bindings are potential bottlenecks in a logic-programmed computer. Accordingly, files stored by the FSD should be kept small in size. However, very large files are commonplace on present-day computer systems. The observation that most large data collections have some form of internal organization (records, rasters, function or subroutine hierarchy, etc.) suggests a solution to this dilemma. Large collections of information can be stored in a number of smaller files, one logical unit per file†. A server can assist by accessing constituent files and emulating the larger view of the information. A user need not be aware of the actual granularity of storage. Any necessary directory or dictionary-type information is also stored in separate files.

Other plausible, useful facilities which could be supplied by servers include data encryption/decryption, access restrictions, read-only files, and UNIX-like directories.

---

† Waterloo Port [Malcolm et al. 83] and Waterloo UNIX Prolog [van Emden & Goebel 84] are examples of an operating system and a language subsystem, respectively, where storage of related information over a hierarchy of small files is the norm.

3. Discussion

The FSD is the formative heart of the file system. It is also the most innovative aspect. This section argues that its implementation is not unreasonable and provides motivation for certain of its characteristics.

3.1. Term-based I/O

Predicate calculus dictates that arguments in an atomic formula be terms. Hence, terms are the basic data construct in logic programs. It seems only proper, then, to consider I/O within a logic inference machine and within logic programs on the basis of terms.

A term is a variable, a constant symbol, or an expression of the form

$$f(t_1, \ldots, t_j)$$

where f is a j-place function symbol and $t_1, \ldots, t_j$ are terms. In most logic programming languages, goals and clauses can also be treated as terms. The Prolog system predicates *clause* and *call* are cases in point.

Computation in a logic inference machine involves the manipulation and interpretation of terms. Certainly, if hardware with such capability can be designed and built, peripheral devices can be developed which accept, supply, or store information on a term basis. For example, the operation of a logic inference machine requires that terms be transferred between main memory and the CPU. As a magnetic disk is another form of memory, it should not be unreasonable to transfer terms to and from nonvolatile storage.

Implementationally, terms are represented by collections of binary digits. It is convenient and customary to impose a higher-order structuring on bits, such as bytes, fields, words, etc. A term may therefore be represented by a collection of bytes or other fixed-sized units. These units may be utilized in physical transfer of the term. However, as a computational step of an inference machine views all the bits representing a term as a single entity, so an I/O operation should involve the entire term.

3.2. Implementation of a File System Device

The FSD is a device process. The exact nature of its supporting hardware (device processor and associated physical I/O device) is not part of the operating system design. However, an implementation should be conceivable. This section explores avenues towards this end.

The idea of a device processor is compatible with the evolution of microprocessors and the trend toward more "intelligent" hardware. This migration of "intelligence" is demonstrated by, for example, ICOT's PSI where unification and many common operating system kernel functions are implemented in firmware [Uchida et al. 83, Yokota et al. 83]. The sophistication necessary to realize a FSD is not unreasonable given such advances in hardware design.

A major concern in implementing a FSD is ensuring that the file system data structure is nonvolatile. Methods of fulfilling this requirement are dependent upon the nature of the CP machine, and in particular, the techniques used to support shared variables and propagate their bindings.

With a number of techniques – a large, multiple-access, global memory, for example – stable storage might be accomplished by periodically checkpointing the process state onto a medium such as a magnetic disk. The operation need not be active on the part

of the device processor. It can be instigated and controlled by a separate microprocessor associated with the physical media. The frequency of the checkpoint operation must ensure reasonable data accuracy, yet not impede the operation of the processor, or the CP machine as a whole. The idea of providing nonvolatile storage by checkpointing is not unknown. In the Eden system [Black 83], files are active entities rather than data structures. A checkpoint primitive is the only kernel mechanism by which files access stable storage.

If the machine architecture is a network of processors, variable bindings may be disseminated by interprocessor "packets" (e.g. Bagel [Shapiro 83b]). In this case it is possible to treat file storage as a database application. The state of the FSD is recorded in the database. Notification of a new variable binding is treated as an update transaction by the device processor. Requests for binding information are handled as database queries.

Another approach is possible in a distributed architecture. The device processor supporting the FSD can be similar to other, nonspecialized processors. However, it must be able to accommodate larger data structures – that of the file system – as part of its process state. It can do so by making use of slower, nonvolatile storage as part of a virtual memory. Caching techniques expedite references to recently accessed files.

The storage and retrieval of information based on the file identifier is not difficult to achieve. In a conventional architecture, information is retrieved from stable storage through a numerical address lying within a predetermined range. In the case of the FSD, the file identifier, a ground term, can be transformed into such an address by applying hashing functions.

Obviously, all of these points are deficient in implementational detail. They do suggest, however, that the construction of a FSD is conceptually plausible.

3.3. Efficiency

For a logic inference machine to be efficient access to shared terms must be efficient, the amount of information that is stored to accommodate large shared data structures must be minimized, and shared variable bindings must be disseminated rapidly. The efficiency of accessing files stored by the FSD is dependent on precisely these considerations. Consequently, the methods used to satisfy them have a direct bearing on the efficiency of the FSD. A similar case may be made for file system servers.

4. Restrictions and Alternatives

As with any computer system component, there are alternatives possible in the design of this file system. Such alternatives are the subject of this section.

As described in Section 2.2, it is desirable to keep files stored by the FSD small in size. Implementationally, it may prove practical to impose size restrictions, both of file names and contents†. Attempting to exceed the set limits would result in a hardware error, probably reported to the user as a goal failure or error-indicating reply message.

The inability to update files stored by the FSD need not be an encumbrance. Section 2.2 suggests one method to conceal the limitation using a server. A list of atoms is supported as the file content. Another possibility is for the server to treat a file as the

---

† This is part of a greater problem with CP, and logic programming languages in general. In theory, it is always possible that some process will construct a term that exceeds the memory capacity available to contain it.

history of a set of data, in the spirit of mutable arrays [Eriksson & Rayner 84]. For example, a file could contain lists of the current and previous values of each logical data element. On update, a new value is prepended to the beginning of the appropriate lists.

An alternative to having the file content as an arbitrary term is requiring it to be a list of terms. The file system device could then support "read" and "write" requests for accessing the constituent terms of a file. However, this alternate approach still relies on a file system data structure stored as a process state and data transfer through unification and shared variables. Therefore, considerations regarding file size and the efficiency of variable binding propagation would be no less important. Further, the two approaches can be seen as comparable by noting that a list of terms is still a term, and that an arbitrary term can be stored as a list of exactly one term.

A significantly different scheme for file storage [Cleary 84] involves turtle annotations for CP [Shapiro 83b]. Turtle notation is extended to allow absolute locations: designations that a process must be executed on a specific processor. Using this idea, files are regarded as active processes created on demand. They modify their states in response to "read" and "write" messages. A "close" message, however, is processed as follows:

file([close|ReqStrm], FileContent) :-
file(ReqStrm?, FileContent)@disk.

That is, the *close* simply directs the process to the processor designated as *disk*. This processor stores a nonvolatile representation of any (file) process which migrates to it for later retrieval and activation. This idea is promising, though many details have yet to be determined.

It is possible to drop the requirement that file identifiers be ground. However, the properties of the resulting file system model have yet to be explored.

5. Summary and Concluding Remarks

This paper has described the file system of an operating system for an abstract logic inference machine. Many aspects of the design are noteworthy. The file system device (FSD), the entity responsible for nonvolatile storage of information, is a special type of name server. The FSD facilitates the creation, access (reading or writing), and removal of files. File contents and file identifiers are arbitrary terms, though identifiers must be ground. A file is open when its content term is being shared between client and FSD. Unification supplants typical data transfer operations. File system servers build on the capabilities of the FSD, and can provide useful features of conventional file systems. This novel and simple view of a file system is due to the power and versatility of CP – and similar languages – and the characteristics of the machine model. Implementation of this file system is not unreasonable given the ability to construct efficient parallel logic inference machines.

An advantage of this approach is that there is no need to introduce new constructs to CP, or make use of novel programming techniques or unique properties of the language. The design is based on first-order logic, without recourse to side effects.

Research into this file system design, as well as that of the overall operating system and CP machine model, is ongoing. A small set of file system servers which seem complementary and particularly useful are being developed. Future papers will elaborate on aspects of the operating system and CP machine model presented here in summary form.

Acknowledgements

Credit is due to Harvey Abramson for his encouragement, suggestions, and receptive ear during the evolution of this paper. Also, John Cleary, Steve Gregory, Nir Friedman, and Colin Mierowsky contributed valuable comments and criticism.

References

[Black 83].
A. P. Black, "An Asymmetric Stream Communication System," in *Proceedings of the Ninth Symposium on Operating Systems Principles*, p. 4-10, ACM, Bretton Woods, New Hampshire, October 10-13, 1983.

[Cheriton 79].
D. R. Cheriton, "Multi-Process Structuring and the Thoth Operating System," Tech. Rep. 79-5, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, March 1979.

[Chikayama 83].
T. Chikayama, "ESP – Extended Self-contained Prolog – as a Preliminary Kernel Language of Fifth Generation Computers," *New Generation Computing*, vol. 1, no. 1, p. 11-24, Tokyo, 1983.

[Clark & Gregory 84].
K. L. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," Research Rep. DOC 84/4, Department of Computing, Imperial College, London, April 1984.

[Cleary 84].
J. G. Cleary. Personal communication.

[Clocksin & Mellish 81].
W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

[Eriksson & Rayner 84].
L.-H. Eriksson and M. Rayner, "Incorporating Mutable Arrays Into Logic Programming," in *Proceedings of the Second International Logic Programming Conference*, ed. S.-Å. Tärnlund, p. 101-114, Uppsala, Sweden, July 2-6, 1984.

[Hattori & Yokoi 83].
T. Hattori and T. Yokoi, "Basic Concepts of the SIM Operating System," *New Generation Computing*, vol. 1, no. 1, p. 81-85, Tokyo, Japan, 1983.

[Lockhart 79].
T. W. Lockhart, "The Design of a Verifiable Operating System Kernel," Tech. Rep. 79-15, Computer Science Department, University of British Columbia, Vancouver, B.C., Canada, November 1979.

[Malcolm et al. 83].
M. Malcolm, B. Bonkowski, G. Stafford, P. Didur, "The Waterloo Port Programming System," Technical Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, January 1983.

[Moto-oka 82].
T. Moto-oka (ed.), *Fifth Generation Computer Systems*, North-Holland, Tokyo, Japan, 1982.

[Ritchie & Thompson 74].
D. M. Ritchie and K. Thompson, "The UNIX Timesharing System," *CACM*, vol. 17, no. 7, p. 365-375, July 1974.

[Shapiro 83a].
E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," TR-003, ICOT – Institute for New Generation Computer Technology, Tokyo, Japan, January 1983. Also available as CS83-06, Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

[Shapiro 83b].
E. Y. Shapiro, "(Lecture Notes on) The Bagel: a Systolic Concurrent Prolog Machine," TM-0031, ICOT – Institute for New Generation Computer Technology, Tokyo, Japan, November 1983.

[Shapiro 83c].
E. Y. Shapiro, "Systems Programming in Concurrent Prolog," TR-034, ICOT – Institute for New Generation Computer Technology, Tokyo, Japan, November 1983.

[Shapiro 84].
E. Y. Shapiro, "Updates from the Weizmann Institute," *Prolog Digest*, vol. 2, no. 24, internetwork mail, June 27, 1984.

[Shapiro & Takeuchi 83].
E. Y. Shapiro and A. Takeuchi, "Object Oriented Programming in Concurrent Prolog," *New Generation Computing*, vol. 1, no. 1, p. 25-48, Tokyo, 1983.

[Takagi et al. 84].
S. Takagi, T. Yokoi, S. Uchida, T. Kurokawa, T. Hattori, T. Chikayama, K. Sakai, J. Tsuji, "Overall Design of SIMPOS," in *Proceedings of the Second International Logic Programming Conference*, ed. S.-Å. Tärnlund, p. 1-12, Uppsala, Sweden, July 2-6, 1984.

[Takeuchi & Furukawa 83].
A. Takeuchi and K. Furukawa, "Interprocess Communication in Concurrent Prolog," in *Proceedings Logic Programming Workshop '83*, p. 171-185, Algarve, Portugal, June 26 - July 1, 1983. Also as ICOT Technical Report TR-006, 1983.

[Uchida et al. 83].
S. Uchida, M. Yokota, A. Yamamoto, K. Taki, and H. Nishikawa, "Outline of the Personal Sequential Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 1, p. 75-79, Tokyo, 1983.

[van Emden & Goebel 84].
M. H. van Emden and R. G. Goebel, "Waterloo Unix Prolog User's Manual," Version 1.2, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1984.

[Yokota et al. 83].
M. Yokota, A. Yamamoto, K. Taki, H. Nishikawa, and S. Uchida, "The Design and Implementation of a Personal Inference Machine: PSI," *New Generation Computing*, vol. 1, no. 2, p. 125-144, Tokyo, 1983.