SOME NOTES ON A DEL BASIS FOR LANGUAGE-ORIENTED OPERATING SYSTEMS

by

Michael J. Flynn

and

Martin Freeman

November 1979

TECHNICAL NOTE NO. 169

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

The work described herein was supported in part by the Army Research Office - Durham under contract no. DAAG29-78-0205 and by the National Science Foundation under grant MCS 76-07682.

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305

Technical Note No. 169

November 1979

SOME NOTES ON A DEL BASIS FOR LANGUAGE-ORIENTED OPERATING SYSTEMS

by

Michael J. Flynn

and

Martin Freeman

ABSTRACT

The concept of a noninterruptable atomic function has been used for describing operating systems. In this note we suggest certain extensions to this model of operating systems for use in language directed architectures (called Directly Executed Languages--DELs). Four categories of functions are defined: absorbed functions, atomic functions, constructed functions and metalingual functions. A possible scenario as to the relationship between this functional partitioning and operating system routines is described.

The work described herein was supported in part by the Army Research Office - Durham under contract no. DAAG29-78-0205 and by the National Science Foundation under grant MCS 76-07682.

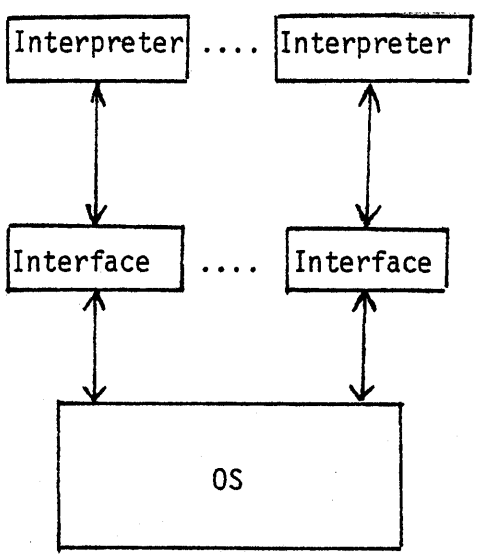
LANGUAGE-ORIENTED OPERATING SYSTEMS

Traditional language interpreters supported by conventional operating systems usually require relatively elaborate interfaces to match the needs of the interpreter with the services supplied by the operating system. In a multi-language environment each language interpreter requires a different interface. The operating system in this case is not directly integrated into the language interpreter (Figure 1a).

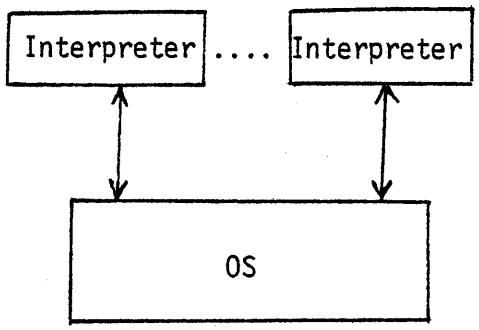
In a language-oriented operating system, however, the choice of system facilities is driven by the given language. In a multi-language environment, the operating system must be flexible enough to provide support for many different language features. For instance, a Fortran-like language would probably make little demand of the operating system, perhaps only as far as I/O operations are concerned; whereas a language like Concurrent Pascal would probably be more demanding--e.g. support of concurrent processes. In any case, the facilities required by each language must be taken into consideration in designing the operating system.

We believe that there are a set of concepts, structures and mechanisms that are common to higher-level languages, and that these elements can be embodied in an operating system which directly supports the (concurrent) operation of different language interpreters [1] (Figure 1b).

A primary concept in developing such a system is that of an atomic function (or action) [1,2,3]---a non-interruptable process providing support for language primitives. For example, the language primitive WRITE in Fortran might have as one of its associated atomic functions the initiation of a channel command. Such language primitives can be realized with sequences of atomic functions; interrupts



(a) Matching Needs to OS



(b) OS Support Influenced by Interpreter Needs

Figure 1

being serviced at atomic boundaries.

In designing and creating atomic functions it is desirable to achieve as sizable an execution time as is possible in order to minimize the relative significance of entry and exit overhead during interpretation. However, peripheral device characteristics require that interrupts must be serviced by a specified time after their occurrence or information will be lost as well as, perhaps, opportunities for faster operation. Thus atomic functions should be selected so that their execution times fit within a fixed time envelope.

A system structured by this approach has several advantages including (1) a more transparent system design (e.g. providing direct support for language primitives), (2) the opportunity for efficient operation by minimizing the number of objects interpreted, and (3) the natural protection mechanisms inherent in an interpretive approach.

DIRECTLY EXECUTED LANGUAGES

Program forms or representations can be created which closely correspond to particular high-level languages. Such forms are called high-level language machines or in our terminology Directly Executed Languages [4,5]. These are intermediate representations of original high-level language programs which are particularly suited for interpretation by a host machine under an interpretive program control. The DEL representation is an intermediate form of the program, it is not the high-level language itself; the DEL is directed at representing HLL objects rather than host objects, however. Thus, A * B → C appears as one DEL instruction, *, and object names A, B, and C are coded with respect to their environment (scope each as a single object),

e.g. consider: A * B + C →D

LD	X=0	A
*	X=0	B
+	X=0	C
STO	X=0	D

(a) Traditional Single Accumulator Representation (X=0 means index unused)

*	A	B
+	C	D

(b) DEL code

Since each higher-level language has its own representation and interpreter, both operations and operands are interpreted as defined by the higher-level language. The instruction vocabulary is exactly the same as the HLL actions. The data types available (interpretive) again correspond to those defined in the HLL. Careful DEL design will avoid the introduction of any object not present in the HLL representation; i.e. no temporaries, etc. In the above, a powerful format set associated with the OPS (operation) allow the result of the * to be a source of the + operation. However the DEL is not the HLL -- values either are or can be readily bound to operand names. Coding of objects is not mnemonic; rather it is very concise. Object container sizes can be

restricted to $\log_2$ of the number of such objects in the HLL scope of definition--usually a significant savings over host oriented identifiers. Notice in the above example the traditional machine includes an index field with each instruction identifier, even though in this particular example it is not used. The DEL identifier on the other hand should be regarded as a concise pointer to a more detailed data descriptor of the object to be used.

Experience with the DELs for FORTRAN show static size improvement of 5 to 1 in required representation space and about 4:1 improvement over the number of instructions to be interpreted when compared with traditional host oriented instruction sets [5].

A simple DEL implementation model is shown in Figure 2. The DEL program representation lies in image store (main memory) while the interpreter and the interpreter parameters lie in a special high speed interpretive storage (corresponding to a read/write microprogram store). The host is unbiased with respect to any image instruction set--i.e. it is a special purpose machine designed for high speed interpretation.

The resulting arrangement is sometimes called soft architecture. The representation to be interpreted depends completely upon the interpreter. Non-dedicated host machines (not dedicated to any particular image) incur some overhead due to their inherent flexibility. However, this is more than compensated for by the smaller number of objects required to be interpreted in a DEL program representation. Moreover, specially dedicated host machines could be defined which would more closely correspond to a particular HLL and its environment, avoiding in this case even this interpretation overhead.

An operating system itself, of course, is an interpreter. It falls naturally into the DEL model. However, the types of associated functions and their use in an operating system require careful consideration of whether particular

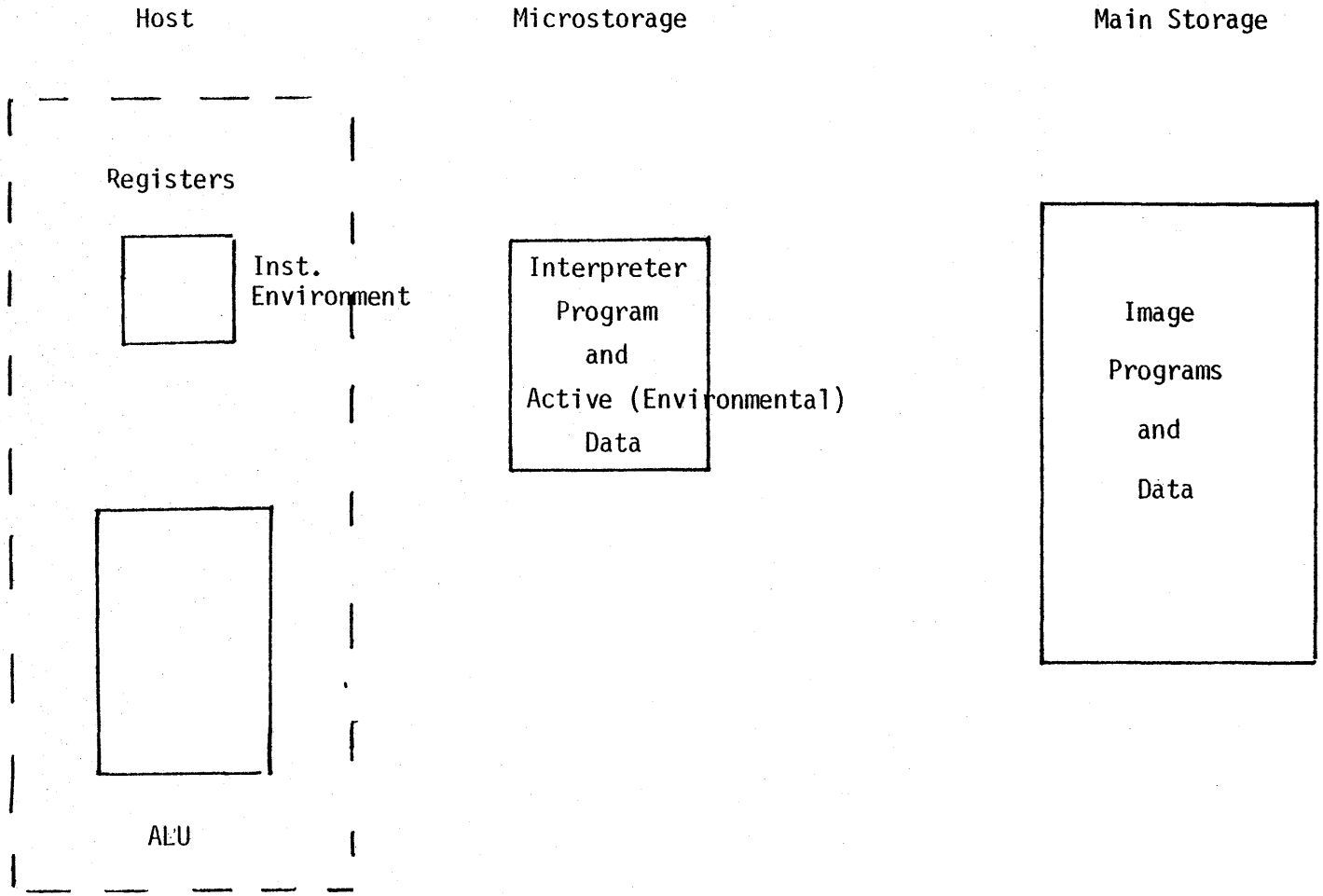


Figure 2. DEL Storage Assignments

operating system actions should be regarded as belonging to the image program representation or to the interpreter itself. This partitioning is of primary importance in the remainder of our discussion.

RELATIONSHIP OF HIGHER-LEVEL LANGUAGES AND OPERATING SYSTEMS

An opportunity exists with a DEL structure based upon atomic functions for a unification of the concepts in higher-level language design and operating system construction. Clearly there exists a DEL for any given higher-level language, and this DEL sits at the interface between the user and the hardware. There can also be a DEL for the operating system which sits at the interface of the language interpreter and the operating system support. This DEL would consist of the atomic functions.

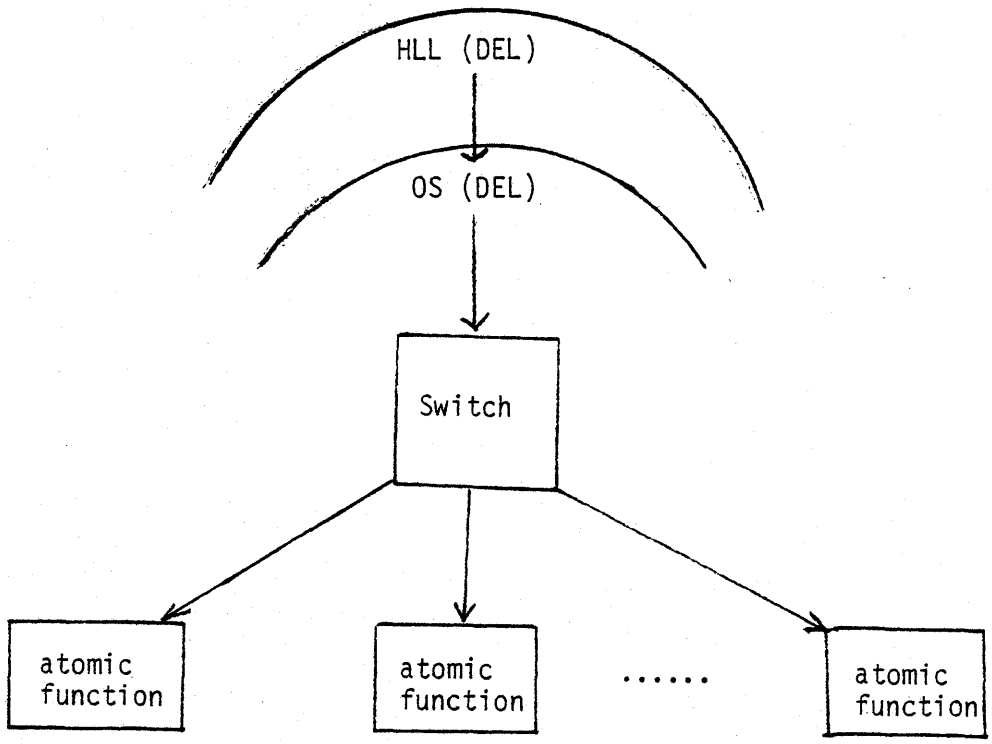


Figure 3. Logical Realization of an OS DEL

At this lower level, there could be a switching mechanism which sequences through the atomic functions associated with a given request from a language primitive. In a sense, the switch is a form of logical control unit and the atomic functions are a form of logical functional units (see Figure 3). This switch may be realized in many ways. It could take the same form as the control for the higher-level language DEL (e.g. dynamic contour) or could take the form of the Supervisor detailed in [6].

Protection

The protection of users and programs is natural in this type of system. As an example, let us assume that the interpretive system is realized in two levels:

1) the language interpreter level (level 2)
2) the OS support level (level 1)

Protection of the address space and procedures of the language is easily imbedded in the language interpreter. In fact, a great deal of protection is afforded in most higher-level languages by the unavailability of the use of the idea of a storage address. A microprogrammable host machine makes it possible for the implementation of protection mechanisms to be efficient.

Protection is also possible in the interface between level 2 and level 1. System support functions can be addressed by capabilities [7]. A capability is a special kind of address for a virtual object that can only be created by the system. In order to use the object the capability provides accessing information. Of course the advantage of a capability scheme is that access to a system facility can be checked without actually handling the system object----i.e. checking can be accomplished entirely at level 2.

Atomic Functions

In the traditional view of architecture, the instruction set is the interface between user programs and the system resources . The instructions are low-level operations interpreted by a control unit activating system resources. These resources--flip-flops, register, memory locations, etc.-- are assumed to always be immediately available and modifiable.

In designing atomic functions for the higher-level instruction set of an operating system, we take a different view since the resources managed are more complex. There are two possible partial characterizations of atomic functions based upon resource allocation:

(1) those requiring explicit resource allocation, and

(2) those requiring implicit resource allocation.

Type (1) implies that there are atomic functions to allocate system resources as well as to transfer objects from one storage medium to another. For example, the code for an atomic function search may be in micro-storage ready for execution but missing needed resources. The search function determines whether an item is a member of a portion of a file. Before we can execute this function we must execute:

(a) allocate (main memory)

(b) transfer (from disk memory to main memory).

Type (2) implies that the system itself provides resources for the associated atomic function before execution, thus obviating the need for explicit allocate and transfer atoms.

The advantage of type (1) functions is that the user has explicit access to system resources and can program more complex function sequences. However, if one considers the allocation of system resources as overhead, then making resource allocation explicit implies that there will be some added system overhead in executing the (overhead) atomic functions (e.g. allocate and

transfer). While flexibility is diminished in type (2) functions, system operation is more efficient----fewer atomic functions are required. Type (2) functions are discussed fully in [6]. We will confine our discussion to type (1) functions however, to allow language interpreters access to the resource allocation mechanisms of the operating system.

Terminology

In the instruction set for a "well mapped" machine organization, instructions are indivisible---i.e. they are non-interruptable and run to completion. Furthermore, instruction sequencing is usually simple---in most cases the next instruction follows in order with a usual limit of three next instructions.

A DEL instruction, on the other hand, is a specification for a higher-level operation. This operation might handle more substantial resources then registers, flip-flops, etc.---e.g. a block of memory instead of a single memory location. Instead of simple branch instructions, CASE type and computed GOTO type operations are possible. Language primitives such as IF and DO are also possible. The increased semantic content gives rise to a set of common atomic functions---the absorbed functions---that can be a part of the realization of language primitives. An example of such a function would be one which maps a logical address into a physical address. This is especially useful in the case where the logical address is an index into the dynamic contour. Another case of an absorbed function might be dynamic storage allocation for an Algol-like language. The implication here is that the storage resources of the underlying operating system are not necessarily used dynamically but that initially the Algol interpreter is assigned a portion of storage from the operating system. From then on the interpreter

manages this storage. Here we are trading storage efficiency for independence (and indirectly speed).

To support a higher-level language interpreter an underlying operating system must be constructed. We feel that a single-layered operating system is not able to efficiently support the multi-lingual, interpretively-oriented situation. We propose a new multi-level model of operating system support in which levels are associated with interpretive functional parameters. We believe that the most variable aspects of an operating system are those most closely related with linguistic operatins and semantics. We therefore propose a four-layered functional hierarchy (see Figure 4):

(1) Absorbed functions

(2) Constituent atoms

(3) Constructed OS functions

(4) Meta-lingual functions.

Unlike absorbed functions, constituent atoms are not a part of a particular language interpreter. They are the atomic functions which provide the essential resources of the operating system to the language intepreter. These services are provided in response to exception conditions and direct calls upon the underlying operating system.

An exception handler can be constructed as a constituent atom. Typical handlers include ones for page faults, memory protection violations, arithmetic exceptions (e.g. overflow), access rights violations, etc.

Certain language primitives cannot be represented as instructions to be handled by the specific language interpreter, they call for services that an operating system must provide. These services should be common to all language interpreters---providing I/O operations, resource allocation, etc.

Constructed OS functions are higher-level primitives composed of constituent atoms. The PRINT command in a language can be a constructed function of constituent atoms which realize the printing operating. Constructed functions are interruptable between constituent atoms.

Meta-lingual functions are instructions that surround a user's program in a batch environment and are commands typed at terminals in a timesharing environment. Typical examples include LIST(file, LOAD(file), (see Figure 5). Meta-lingual functions are composed of constructed OS functions and constituent OS atoms.

It is in the meta-lingual functional definition that the OS designer first becomes the language designer. Conciseness, straight-forwardness and usefulness of representation are hallmarks of successful meta-lingual command design. Even at this level, however, flexibility is possible so that meta-lingual functions (commands) may have alterable definitions depending upon the environment.

Figure 6 shows the levels of OS function assignment in an interpretive processor. The absorbed functions naturally lie within the language interpreter itself. The routines for commonly used constituent atoms also reside in the interpretive storage, while constructed OS functions written in terms of constituent atoms and requiring a dual level of interpretation may lie in either the interpretive storage or in the image storage. Meta-lingual functions, which will surely consist of special constituent atomes as well as constructed functions, reside outside program storage until required.

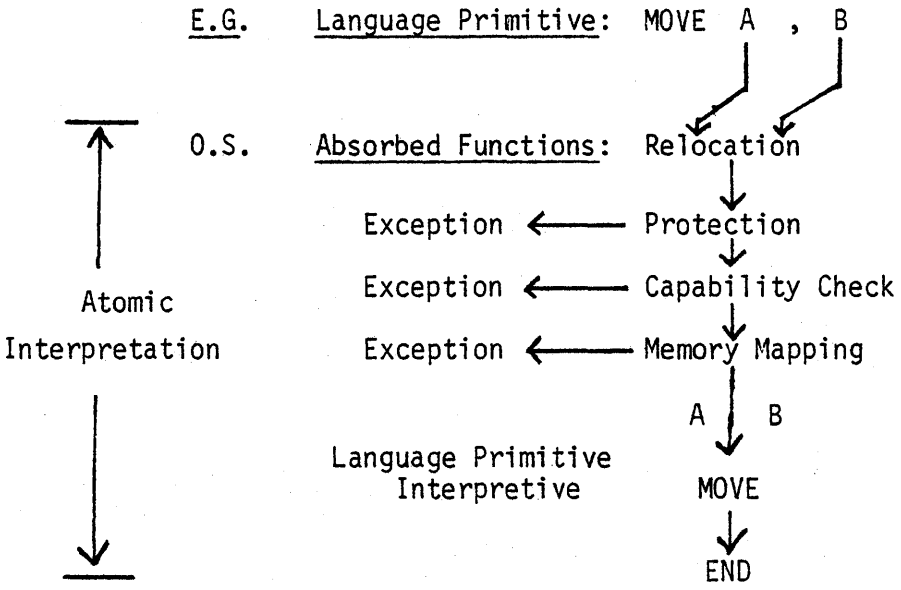
To place in prespective the ideas just presented, let us consider a typical situation where these ideas may be employed. Consider an Algol-based system about to execute a user routine (written in Algol) involveing I/O operations via the meta-lingual function EXECUTE (routine). Let us view the overhead functions that must come into play for it to start executing. Table 1a shows

the steps involved prior to execution. These steps are not necessarily in order. Table 1b, on the other hand, categorizes events occurring during the execution of the routine according to the type of atomic function called upon to deal with the events.

CONCLUSIONS

The atomic model of operating system processes is a useful means of incorporating directly executed languages into operating systems requirements. The model recognizes that either the semantics of a lingual action (language primitive) or the maximum time for interrupt exclusion will limit functional definitions and hences form a uniform basis for partitioning functions. The model, while attractive, cannot be accessed as to value until several implementations are realized. This note describes the conceptual framework preliminary to actual implementations.

(1) Absorbed Functions: OS functions that are short enough to fit within the interpretation of a language primitive



(2) Constituent OS atoms: OS functions that lie outside the language primitives, yet can be interpreted within atomic time requirements.

E.G. Capability Exception
Page Fault Handler
I/O Buffer Allocation

(3) Constructed Functions: Language primitives which cannot be interpreted in atomic time requirements

E.G. Read
Write

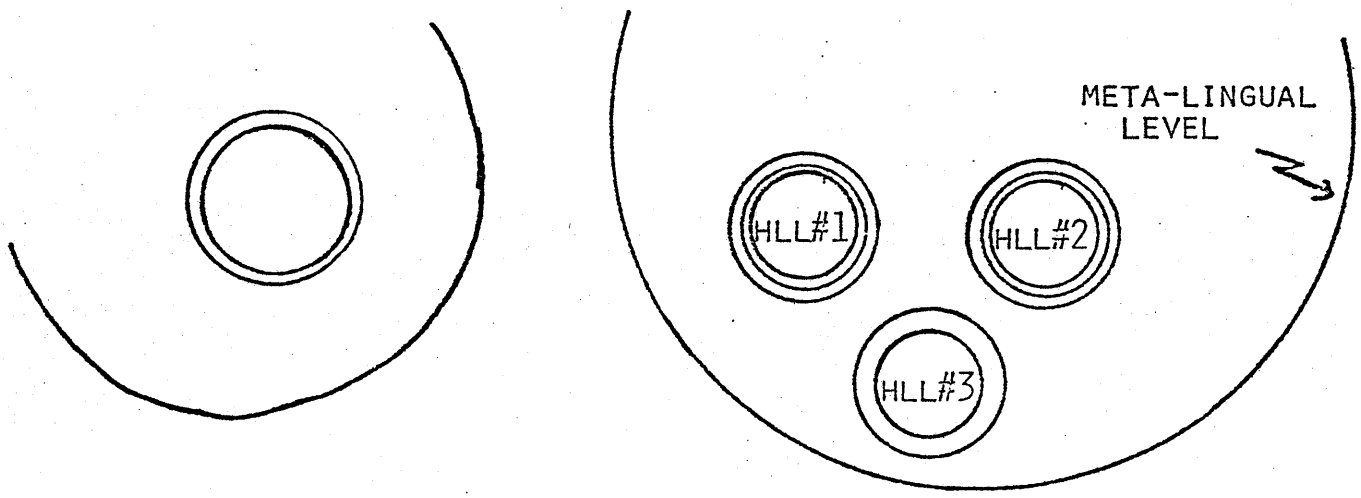
(4) Meta Lingual Functions: Constructed functions that are system primitives and that lie outside any single language.

Figure 4. O.S. Interpretive Hierarchy

META LINGUAL FUNCTIONS

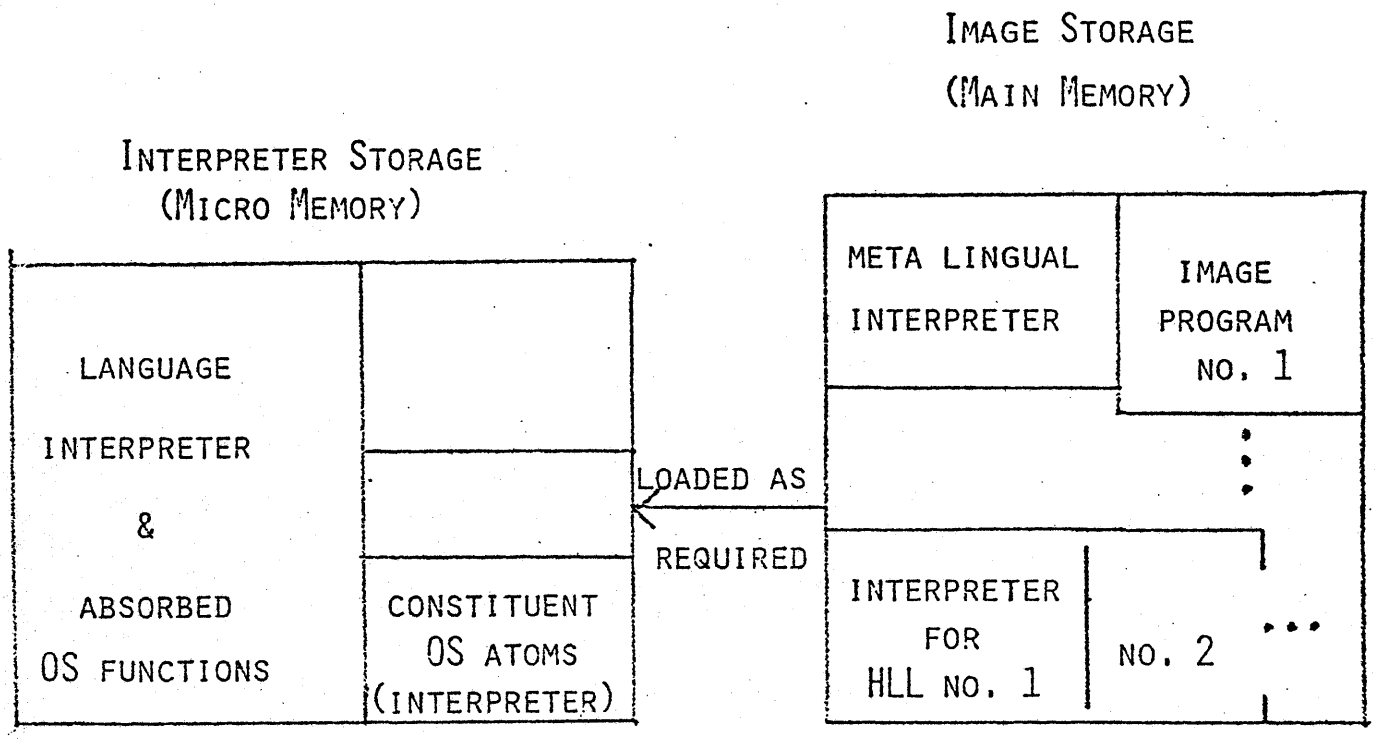
FUNCTION	PARAMETERS	SEMANTICS
CLOSE	⟨FILE⟩	RESIDENT PORTIONS OF FILE RELEASED FROM MAIN MEMORY ALONG WITH ASSOCIATED SYSTEM INFORMATION.
COPY	⟨DESTINATION FILE⟩⟨SOURCE FILE⟩	COPY ⟨SOURCE FILE⟩& CALL IT ⟨DISTINATION FILE⟩.
CREATE	⟨FILE⟩	ALLOCATE APPROPRIATE SECONDARY STORAGE FOR ⟨FILE⟩ AND MAKE SYSTEM ENTRIES FOR ⟨FILE⟩.
DELETE	⟨FILE⟩	DELETE ⟨FILE⟩ FROM THE CURRENT DIRECTORY OF FILES.
RUN	⟨CODE FILE⟩⟨INPUT FILE⟩⟨OUTPUT FILE⟩	RUN THE PROGRAM IN ⟨CODE FILE⟩ WITH INPUT FROM ⟨INPUT FILE⟩ AND OUTPUT TO ⟨OUTPUT FILE⟩ .
LIST	⟨FILE⟩	LIST THE CONTENTS OF ⟨FILE⟩.
LOAD	⟨FILE⟩	BRING ⟨FILE⟩ INTO MAIN MEMORY.
MERGE	⟨FILE 1⟩, ⟨FILE 2⟩	⟨FILE 1⟩ AND ⟨FILE 2⟩ ARE APPENDED TOGETHER AND BECOME THE NEW ⟨FILE 1⟩.
OPEN	⟨FILE⟩ , ⟨ACCESS⟩	IF ⟨ACCESS⟩ TYPE IS PROPER, OBTAIN PHYSICAL ADDRESS OF ⟨FILE⟩ AND CONDITION MAIN MEMORY FOR ⟨FILE⟩ ACCESS.

Figure 5



LAYERS OF INTERPRETATION IN AN OPERATING SYSTEM

Figure 6a



STORAGE ASSIGNMENTS

Figure 6b

EXAMPLE:

Given a routine written in ALGOL involving I/O which is located on secondary storage in object code form, what overhead functions must come into play for the routine to start executing.

EXECUTE Routine

Steps Involved Prior to Execution	Semantics
Find (Meta Lingual)	Determine address of routine on secondary store
Bring (Meta Lingual)	Transfer code to image store
Migrate (Constituent Atoms)	Transfer required constituent atoms to microstore
Setup (Constituent Atoms)	Initialize buffer allocation, type of buffering, etc.
Initiate (Meta Lingual)	Start program running

Table 1a

Events During Execution	Elements Handling These Events
Allocation of I/O Buffers	Constituent atom
Traps (overflow, memory protect, etc)	Absorbed function
Termination	———
READ or WRITE language primitive	Constructed OS function
Page Fault	Constituent atom
Dynamic Memory Allocation in Routine	Absorbed function
Resumption from I/O wait	Constituent atom

Table 1b

REFERENCES

1. Freeman, M., Jacobs, W. W., and Levy, L. S. "On the Construction of Interactive Systems," Proceedings of the 1978 National Computer Conference, June 1978, pp. 555-562.

2. Jacobs, W. W., "Control Systems in Robots," Proceedings of the ACM 25th Anniversary Conference, Vol. 1, 1972, pp. 110-117.

3. Lomet, D. B. "Process Structuring, Synchronization and Recovery Using Atomic Actions," Proceedings of ACM Conference on Languague Design for Reliable Software, SIGPLAN Notices 12, 3 March 1977, pp. 128-137.

4. Flynn, M. J., "The Interpretive Interface: Resources and Program Representaton in Computer Organization," Proceedings of the Symposium on High Speed Computers and Algorithms, April 1977, (D Kuck, et al, Ed.) Academic Press (Pub.).

5. Flynn, M. J., "Computer Organizational Architecture," Operating Systems, (Bayer, et. al., Ed.) No. 60 in Lecture Notes in Computer Science, Springer-Verlag, 1978, pp. 18-98.

6. Freeman, Martin, Jacobs, W. W., and Levy, L. S., "A Model for the Construction of Operating Systems," Proceedings of the 1978 Johns Hopkins Conference on Information Sciences and Systems, April 1978.

7. Dennis, J. and van Horn, E., "Programming Semantics for Multiprogrammed Computations," CACM 9, 3 (March 1966), pp. 143-155.