AN EXPERIMENTAL IMPLEMENTATION OF THE KERNEL/DOMAIN ARCHITECTURE

MICHAEL J. SPIER

THOMAS N. HASTINGS   DAVID N. CUTLER

Department of Software Engineering[1]
Digital Equipment Corporation
Maynard, Massachusetts 01754

ABSTRACT

As part of its effort to periodically investigate various new promising concepts and techniques, the Digital Equipment Corporation has sponsored a research project whose purpose it was to effect a limited implementation of a protective operating system framework, based on the kernel/domain architecture which has increasingly been propounded in recent years.

The project was carried out in 1972, and its successful completion has led to a substantial number of observations and insights. This paper reports on the more significant ones, specifically: 1) the techniques used in mapping a conceptual model onto commercially available hardware (the PDP-11/45 mini-computer), 2) the domain's memory mapping properties, and their impact on programming language storage-class semantics, 3) this architecture's impact on the apparent simplification of various traditionally-complex operating systems monitor functions, and 4) the promise this architecture holds in terms of increased functional flexibility for future-generation geodesic operating systems.

Keywords: Address Space, Domain, Domain Incarnation, Kernel, Memory Space, Process, Protection.

CR Categories: 4.20 4.30 6.20

"...today, we tend to go on for years with tremendous investments to find that the [computer] system, which was not well understood to begin with, does not work as anticipated. We build systems like the Wright Brothers built airplanes - build the whole thing, push it off a cliff, let it crash, and start all over again".

R. M. Graham [N1]

1. INTRODUCTION

Few professionals fail to acknowledge the fact that our software industry is going through what might be described in terms ranging from the mild "difficulty" to the emphatic "crisis". For a comprehensive insight into this problematic state, we refer the reader to the two NATO conference reports [N1,N2] where panels of top professional authorities express their deep-felt concern.

[1]This paper reports on a pure research project sponsored by the Department of Software Engineering, and may not be construed to imply any product commitment by the Digital Equipment Corporation.

Opinions are divided when it comes to pinpointing specific sources from which new techniques might emanate, to alleviate and redress the software situation. There exists a certain gulf between Academe's computer scientists and the industry's software workshops. One of industry's prevailing claims is that the software conceptualist has little grasp on the economic realities of the business. For the opposing view, Barton [B1] contends that the software situation is a self-inflicted positive feedback state created by the industry. Concerning the industry's great reluctance to change its *modus operandi*, Dijkstra is attributed, in [N1], with the remark about "puritans falling in love with their own misery".

Be the reason for the problem what it may, the fact remains that today's software becomes increasingly costly to produce, nigh impossible to certify as to correctness, and guaranteed to generate an indefinite expenditure in maintenance costs. Dramatically confronted with the opposing trend for ever more sophisticated, and at the same time less expensive, hardware technology, the problem manifests itself in the light of increased urgency. The kernel/domain hardware architecture whose (software simulated) implementation is described in this paper may provide a hardware/software cost-effective tradeoff to improve software modularity and reliability [S4].

Multiple-user operating systems are especially susceptible to software-quality problems. In an attempt to satisfy the market's ever growing demand for various sophistications, more and more code is being crammed into already exceedingly complex monolithic operating system monitors (i.e., supervisors, executives) to the point of rendering them virtually impossible to maintain at an acceptable level of correctness. The industry is well aware of this fact, and every change -- no matter how trivial -- is typically cautiously debated and evaluated as to whether or not it really has to be incorporated; its implementation is typically undertaken with great reluctance.

When one fixes a bug at some locality A, one is very likely to provoke the spontaneous eruption of other bugs at some localities X,Y,Z. Indeed, major software products (e.g., monitors, compilers) are known to display a rationale-defying -- dynamically balanced -- constant level of bugs, regardless of the amount of corrective maintenance effort expended. Namely, specific bugs get fixed, others spontaneously pop into existence, and the totality of bugs remains a fairly stable phenomenon. Hopkins [N1] has the following facetious "theory" to explain away IBM-OS/360's constant number of approximately 1000 bugs per release: "... IBM have a large stock of new [software] facilities back in Poughkeepsie waiting to be added to OS. Whenever the bug rate falls below 1000 per release, which seems to be about the user toleration level, they put in a few more facilities which has the effect of keeping the bug rate constant". A companion paper [S4] discusses the stable-number-of-bugs phenomenon in greater detail.

Here at the Digital Equipment Corporation, we are aware of the software situation and are always on the lookout for possible steps to be undertaken which might give us insight into the problem's nature as well as suggest possible remedies. It is generally acknowledged that the implementation of software according to the principles of structured modularity as propounded in the literature [D5,D6,D7,E1,P1,W1] presents the right approach. It is also recognized that modularity must be confined to a protective run-time environment where the misdemeanor of module A is guaranteed never to affect the existing state of correctness of other modules (e.g., X,Y,Z above).

In such a protective environment each procedure may be encapsulated within a dedicated memory space which is named domain. The concept, as well as a model mechanism in which the concept's implementability is demonstrated, was earlier developed by Spier [S3]. Another key concept, developed by Spooner [S7], is that of the kernel which is the most highly privileged domain in the system, containing the system's most critically sensitive functions. In current systems, these functions reside within the memory space of many lesser privileged services. Thus, a trivial modification to some peripheral supervisory function, say the modification of the I/O service routines to handle communication interfaces for an experimental terminal, may result in the clobbering of the system's process scheduling table, to the detriment of the entire user community. Moreover, if this bug is related to the real-time interrupt handling of the new terminal, it will manifest itself in a most capricious and non-reproducible way, a problem with which we are all too familiar.

The introduction of much finer levels of protective resolution is expected to have a beneficial effect on program modularity and reliability by 1) removing a protected module's information from the access range of all other modules, thus making it impervious to external direct interference, and 2) forcing logically-independent functions to interact via formal invocations (call/return) only; for a more detailed discussion see [S4].

2. COMPENDIUM

In 1972, the department of Software Engineering undertook the sponsoring of a limited-objective exploratory implementation of such a modularly protective system framework. It was a pure research project whose goals were to attempt the translation of theoretical concepts onto the reality of existing hardware, evaluate their viability and assess their larger implications. Past experience had shown that radical changes in technology tend to have implications which are more often than not inobvious at the conceptual level. A pertinent example is that of demand paging; initially regarded as merely an elegant way with which to circumvent the tedium of memory swapping and overlays, its presently-known implications range from Denning's working set model [D1] to TENEX's access controlled paging machine [B2].

Our intent was not so much the fabrication of a full-fledged operating system, as the development of a sufficient body of operational code to allow us to judge the overall properties of this protective systems architecture, as well as to form some educated guesses concerning its implications. The project was successfully carried out, and yielded enough interesting observations and insights to warrant the publication of this report. To quickly review some of the insight gained, we have observed that:

2.1 A modularly-structured system, consisting of many independently protected small modules, would indeed drastically reduce (if not eliminate all together) the "stable number of bugs" phenomenon.

2.2 Our protective environment was software implemented. A highly modular system would be hopelessly inefficient in such an interpretive environment. It thus seems that the viability of the kernel/domain systems architecture would largely depend on the availability of dedicated supporting protective hardware, such as proposed in [S3], where an inter-domain call is just as efficient (or inefficient) as a common intra-domain call.

2.3 Writing programs for a domain environment, in which memory is mapped in some mighty peculiar ways (as shown later on in this paper), calls for storage class semantics which current high-level programming languages do not provide.

2.4 Traditional concepts, such as process, take on entirely new meaning. We had to assign our own interpretation to certain well established concepts, as well as invent one of our own, named domain incarnation[2].

2.5 Certain traditionally complex operating system functions may be implemented in deceptively simple ways on this new architecture, suggesting that their traditional notoriety for complexity is not intrinsic but rather a function of their unsuitable implementation environment. To illustrate, we shall present our proposal for a highly flexible CPU scheduler whose implementation is surprisingly simple-minded.

2.6 The main reason why we typically implement a single logical operating system on a single dedicated hardware base is because all supervisory code must reside within the protected monitor memory space. We do not claim that it would be impossible to implement two or more concurrent operating systems of disparate nature (e.g., TENEX [B2], THE [D5], MULTICS [O1] and HYDRA [W2]) on a single hardware machine. The fact that it is not customarily done strongly suggests a certain threshold of complexity which practically delimits the functional capabilities of the monolithic monitor. On our proposed architecture with its many monitor-like domains we envisioned the possibility of supporting the concurrent existence of similiar-purpose redundant supervisory services (e.g., 3 file systems, 7 file nomenclature hierarchies, 4 login responders, etc.) which, for all practical intents and purposes, would provide the external appearance of concurrent operating systems of disparate natures. The hardware base would be hidden by the necessary abstractions provided by the kernel domain (and perhaps additional intermediary service domains) along the lines of structured levels of implementation as promulgated by Dijkstra [D7]. See Spooner [S7] for additional detail.

The above points strongly suggest that the kernel/domain architecture has distinct beneficial effects on the resultant operating system, effects which may well propagate themselves to an as yet unsuspected degree. This in turn gives rise to the hope that this architecture may indeed form the foundation for future *geodesic operating systems*[3], and help put the software industry on a sounder footing.

It is the purpose of this paper to report on the results of our research project. Obvious restrictions pertaining to size preclude our ability to present it in a truly comprehensive detailed manner. We have therefore chosen to concentrate on a limited number of issues which would best illustrate the techniques used, and which would convey some of the properties of this novel architecture. Thus, this paper describes in detail only the memory mapping techniques which we developed for domain implementation. This, in our opinion, is the most important technical development to come out of this research project. Companion papers describe how domains may be put to use [S4,S6]. We also present in this paper certain ideas which we would have liked to implement, but did not; thus their presentation is not detailed, rather more in the nature of suggestions for further research work.

[2]The term "domain incarnation" was coined by Thomas N. Hastings of the Digital Equipment Corporation.

[3]From Webster's 7th New Collegiate Dictionary:
Geodesic adj: made of light straight structural elements largely in tension (a geodesic dome)

"We assume throughout our architecture that no software is 100% debugged. When some item appears to be perfect, all we know is that the data applied to it so far has not been known to upset it..... We cater for imperfection by honeycombing the system with [*protective*] fire walls..."

C. R. Spooner [*S7*]

3. BACKGROUND

The idea that the binary all-or-nothing monitor vs. user protective interface might be generalized into an arbitrary number of such protective relationships has been around for quite some time. The Evans & Leclerc parameter space model [E2] provides an important milestone in its development. The idea was similiarly formalized by Dennis & Van Horn in the sphere of protection model [D3], whose C-lists define inviolable address space specifications. Additional work on this concept, notably by (in alphabetical order) Feustel [F2], Graham & Denning [G2], Hoffman [H2], Lampson [L1], Schroeder [S2], Spier [S3], Spooner [S7], Vanderbilt [V1] and Wulf [W2] convincingly demonstrates the idea's viability and implementability (as well as its increased popularity). Much of the above work was motivated by the computer utility concept as proposed by Dennis [D4] and Fano [F1].

From the industry's commercial point of view, the computer utility seems like a laudable futuristic panacea whose present market appeal is at best very limited. Thus, the inseparably-related protective issue is all too often considered as an interesting academic extrapolation which might not, however, be worth the substantial R&D investment required for its development into a marketable product [S4]. Graham [G1] recognized the need for protective fire walls to sharply delimit the propagation of programming errors, and suggested an hierarchical organization of nested protective environments which he named rings of protection. These rings, whose protective property is well defined only within the context of a single user process, were subsequently implemented on the Multics system [O1,S1]. Their usefulness seems rather limited [G1,S1,S2].

A protective domain system supports two inherently conflicting functions: 1) the guaranteed protection of selected information behind inviolable fire walls, and 2) the controlled breaching of the protective wall to provide selected co-users of the system with carefully defined computational access to the protected information [S3]. And while the first function of absolute protection has been known and used in operating systems for a long time[4], it was the computer-utility which inspired the need for very flexible (and foolproof) mechanisms to support the controlled breaching of protective walls.

It is our opinion that, in view of the implementability of protective domains [S3], one might well put the concept to immediate use on the kind of operating systems that we build today. The more complex protective mechanisms which were developed with the computer utility in view, support not only very flexible means for the controlled breaching of protection, but also elaborate ways in which access authorizations may dynamically be granted and revoked, even traded among co-users of the system. If we decide not to support these more esoteric refinements, restricting ourselves to some simple scheme for the controlled breaching of protection, sufficient to enable the free flow of control among protected modules, then we are dealing with a somewhat simpler flavor of domain whose implementation complexity is drastically reduced. Given the general nature of our experimental system (i.e., "framework" as distinct from "application") we chose not to develop any specific access control philosophies which we believe to be application-dependent [S3]. Thus, although access was checked at the appropriate logical places, all access authorizations were of the public "to whom it may concern" variety. This paper will therefore go no further into the specifics of access right manipulations, which were extensively covered elsewhere [S3].

We have thus decided to essentially ignore the computer utility with its enterprising users and their commercial interactions [V1], ignore the possibility for having to support mutually suspicious sub-systems [S2], and concentrate on the immediate benefits promised in terms of more modular, less error-prone operating systems. Our view coincides with that of Spooner [S7] and provides, in our opinion, a sufficiently substantiating reason to encourage the industry into further development of such protective systems [S4].

Our research effort was modest and used very modest tools. For our implementation hardware base we chose the Digital Equipment Corporation's PDP-11/45 minicomputer [P2] which features an optional protective hardware device consisting of: 1) three hardware implemented rings of protection (known, in decreasing order of privilege, as kernel mode, supervisor mode, and user mode), and 2) a dual set of protective memory space mapping registers, named active page registers (ARP's), which are not unlike the descriptors proposed for Spier's model mechanism [S3]. We made use of the kernel mode environment in order to implement an interpretive software domain machine whose responsibility it was 1) to correctly map individual domains through usage of the ARP's, and 2) to support the inter-domain call/return mechanism [S6]. The remaining system code resided in user mode domains. No attempt was made to put the intermediary supervisor mode to any use whatsoever.

[4] In a typical n-user process system there exist (n+1) protected memory spaces, namely those of the n user processes plus that of the monitor. And while the monitor has access privileges to anything in the system, a user process is inherently incapable of ever formulating a hardware address which would result in reference to another user's memory space, or to the monitor's memory space.

The reader might find it worthwhile to investigate the implementation details of the HYDRA system [W2], which uses a related hardware base (the C.mmp deriviative of the PDP-11/45) to implement a similiar protective system framework which is based on a parallel model (Dennis's capabilities [D3]).

4. MEMORY- AND ADDRESS-SPACE

Let us define the term memory space to designate the collection of physical memory addresses whose usage would result in actual access of a memory cell. On a computer equipped with m discrete, consecutively addressable memory cells, the hardware memory space would span the range Ø through 2^m-1.

Let us define the term address space to designate the collection of hardware addresses that may be formulated by an executing procedure. On a computer with an n-bit address field, the procedure's address space would span the range Ø through 2^n-1.

Through usage of a hardware address translation device, typically a base/boundary relocation register, we separate the concepts of memory space and address space. Even though an executing computation is still capable of formulating the entire range of potential addresses Ø through 2^n-1, each address is relocated by adding to it the register's base value, and restricted to the actual current memory allocation through enforcement of the register's boundary value. On such a system equipped with relocation hardware, we say that each computation now has a local independent address space, mapped (i.e., forming a window) over an externally allocated memory space. A formulated address i is meaningful only within the computation to which it is local (compare this to the distinction between variable allocation and variable name).

If the relocation register values corresponding to several computations are each mapped over its exclusive memory space partition, then these computations are said to be protected from one another, because none of them could ever formulate an address resulting in effective access to any but its own allocated memory partition.

Evidently, such protection would be hard to enforce if a computation has the ability to set its own base/boundary relocation register values. This register is therefore protected from the computation's power of accessibility, and is exclusively available to the system's monitor only. The monitor is protected from all other computations in the system (because of their inability to ever formulate an address that would result in effective access to the monitor's information), yet by virtue of its ability to load arbitrary values into the relocation register, the monitor has unrestricted access to the hardware machine's entire memory space.

The monitor also differs from all other protected computations in the system in that any such computation may cause its sequence of execution to invade the monitor's memory space, through usage of a programmed interrupt (or trap, fault). The point of entry is a predesignated locality known as interrupt vector element.

5. THE DOMAIN

The protective domain is an independent local address space which is mapped over (partially) exclusive memory space partitions. Similiar to the classical monitor, it may be entered at a predesignated locality known as gate. A computation executing in some domain A may perform an inter-domain call into some domain B there to execute under control of a procedure encapsulated within domain B; the call's target is an approved (and protectable) gate of domain B. The domain is thus a generalization of the classical monitor, best visualized if we think of the inter-domain call in terms of "programmed interrupt", and of the gate in terms of "interrupt vector element".

A processor[5] executing in some domain may thus invade the otherwise inviolable memory space of another domain for the purpose of executing some procedure belonging to the entered domain. The behavior of a procedure may be determined only if entered at an approved entry point; a procedure entered at some random locality is guaranteed to execute unpredictably. Thus, in a domain, system procedure entry points, and especially gates, are protectable items. Building on Spier's model mechanism [S3], we shall not attempt to develop further the protective issue (i.e., manipulation and enforcement of ownership and computational access privileges); taking the domain concept for granted and accepting the implementability and proper protective behavior of domains, we shall concentrate on the memory mapping techniques developed during our actual implementation.

5.1 The Kernel

On the classical non-domain machine, all supervisory code resides in a single monolithic monitor, because the monitor is the only protected memory space which is accessible (via programmed interrupts) to all other computations in the system. Some of the monitor's code is highly sensitive in the sense that

[5]At this point we prefer to use the term processor rather than process, as will be discussed further on. We think of the processor as being an abstract execution agent [S3,S5] acting in behalf of its current procedure.

interference with it would be to the detriment of the entire system (e.g., destruction of the CPU scheduler's table, or of the pager's memory maps would cause what is commonly known as a "fatal system crash"). Other monitor code may be less critically important (e.g., destruction of a per-process data base may perhaps cause no more than the termination of that single user process).

As argued by Spooner [S7] and Spier [S4], overall system reliability could drastically be improved through the enforcement of much finer levels of protective resolution within the monolithic monitor, in order to segregate supervisory functions of disparate critical importance. This is impossible to do on the classical hardware because even less critically important modules still have to be protected from user-mode computations, hence mandatorily are put into the single monolithic monitor. The _rings of protection_, originally developed for this purpose, do not live up to sophisticated expectations.

On the domain machine, which offers an arbitrary number of monitor-like protected memory spaces, it now becomes possible to break up the monolithic monitor into a number of _supervisory domains_, each encapsulating a distinct supervisory function (e.g., macro scheduler, file system, file nomenclature, system control package, login responder, etc.). The supervisory domain containing the lowest-level, most critical system code is named _kernel_. It is in charge of basic system resource control (e.g., CPU, memory, interrupts). Because of its critical nature, the kernel code must be as reliable and free of bugs as is humanly possible. This goal would strongly be compromised if the kernel were periodically to be reopened for modification, for example when some application code evolves and needs new or modified services.

In designing the kernel of our experimental system, we insisted that it be completely application independent and devoid of any decision-making code. We think of the kernel as being a general purpose subroutine performing certain well defined algorithms on caller-supplied arguments. In its capacity as decision maker the caller supplies the argument values, in its capacity as servant the kernel acts on the supplied arguments. Usage of access control [S3] may restrict the availability of certain privileged functions (e.g., disk I/O) to certain trustworthy supervisory domains (e.g., file system domain).

5.2 The Implemented Domain

We chose to base our implementation approach on the Evans and Leclerc _parameter space_ model [E2]. Accordingly, every system data base was associated with a dedicated _caretaker_ procedure, and together with the necessary repositories for run-time information (e.g., local variables) was encapsulated within a dedicated domain. Thus a domain always consisted of a procedure which was specially tailored to manipulate a single logical collection of information items; e.g., a _login manager_ domain consisting of a _login_ procedure and the system's _password table_.

In a system where every procedure resides in its dedicated domain, it is impossible to restrict processor access to the domain on a mutual exclusion basis (i.e., construct the gates in such a way that only one processor at a time may enter a domain) because the situation where two independent processors concurrently attempt the inter-domain call sequences $\underline{A} \rightarrow \underline{B} \rightarrow \underline{C}$ and $\underline{C} \rightarrow \underline{B} \rightarrow \underline{A}$ will cause the system to freeze in a deadlock [H1].

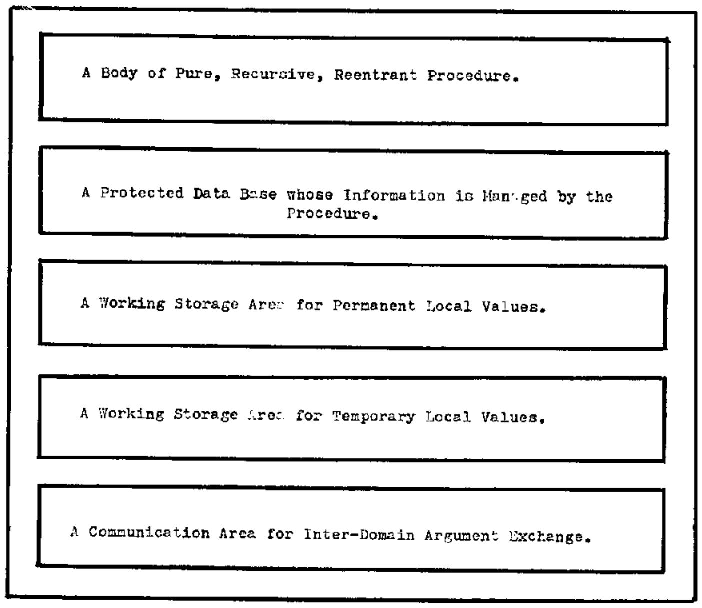

A Body of Pure, Recursive, Reentrant Procedure.
A Protected Data Base whose Information is Managed by the Procedure.
A Working Storage Area for Permanent Local Values.
A Working Storage Area for Temporary Local Values.
A Communication Area for Inter-Domain Argument Exchange.

Figure 1: The Domain's Five Components, Encapsulated within their Protective Wall.

It follows that domains must be capable of supporting the concurrent execution of two or more processors, hence the encapsulated program must be implemented as _pure reentrant_ procedure (assuming, of course, our reluctance to support a proliferation of redundant impure-procedure copies).

A domain as implemented on our system would typically consist of 1) the data base, 2) pure non-self modifying procedure, and 3) local temporary variables allocated to each instance of _procedure activation by a single processor_. In fact, two additional components of a domain were needed, these being 4) a per-processor repository for permanent local storage which is preserved from one procedure activation to another, and finally 5) a data base which is temporarily shared among domains which are connected by a thread of inter-domain calls. This last item is necessary for the orderly passing of arguments across domain boundaries, and corresponds to our way of implementing the _argument domain_ proposed in [S3]. These five components of the domain collectively form the domain's _address space_, as illustrated in Figure 1.

6. THE DOMAIN INCARNATION

If we choose to think of process in its generally accepted definition of processor executing within a memory space [S5], then in a domain system where the execution of a single user job implies the constant switching of memory spaces (read domains) the concept loses its traditional connotation of dedicated user job. Traditional systems used to lump together the concepts of processor and memory space, which in those systems happened to overlap nicely, designating the collective entity "user process". In a domain environment, processor and memory space no longer overlap but are mapped onto one another in complex ways dictated by various access control requirements. Had we chosen to employ the term process in its literal meaning of processor executing within a single domain, we would have had to invent a novel term to designate the larger sequentiality of which a user job consists. As it is, we have chosen for the sake of conformity to stick with the term's traditional meaning of user job.

The concurrent execution of a single domain by several processes implies that if we were to take a snapshot of that domain, we would observe several loci of control [D3], each associated with distinct states of its local variables, each arrested at a distinct stage of progression. We call this manifestation of a single domain, as it is being executed by a single process, domain incarnation. It is analogous to the concept of a reentrant procedure's activation record [J1]. Figure 2 shows a matrix of m processes and n domains, where the intersections represent domain incarnations. If we use numerical identifiers (1,2,3,...,m) for processes and alphabetical identifiers (A,B,C,....,n) for domains, then the execution of domain C by process 4 would be known as domain incarnation C4.

We have earlier mentioned the five address-space components of our implemented domain. We were not more specific because the nature of these components is intimately related to the domain incarnation concept. Each component, which we chose to name segment [D2], is characterized by its peculiar memory mapping and access control attributes, as shown in the list below. When we say that a segment "belongs" to some entity (e.g., domain, process, domain incarnation) we mean that the actual memory partition is allocated/de-allocated at that entity's creation/termination times, respectively. When we say that a segment is accessible to "all processes" we mean that all processes which are authorized to call into some domain may access the segment from within that domain. When we say that a segment is shared by "all domains" we mean that all domains "visited" by some process may access that segment via execution of that very same process. A segment accessible to a domain incarnation only is available to a specific domain/process intersection exclusively; e.g., unavailable to the same domain when executed by a different process, or to the same process when executing within some different domain.

6.1 The Procedure Segment which is an execute-only body of pure, non-self modifying code, whose single physical copy is shared by all processes.

6.2 The Domain Own Segment which is the readable/writable data base manipulated by the procedure, and whose single physical copy is shared by all processes. It is characterized by the fact that it may only be modified by the procedure under conditions of mutual exclusion (i.e., locking).

6.3 The Incarnation Own Temporary Segment which contains the process's execution stack and other temporary local variables. It is by definition unsharable among either processes or domains, hence it is mapped over as many physical copies as there are processes executing within that domain.

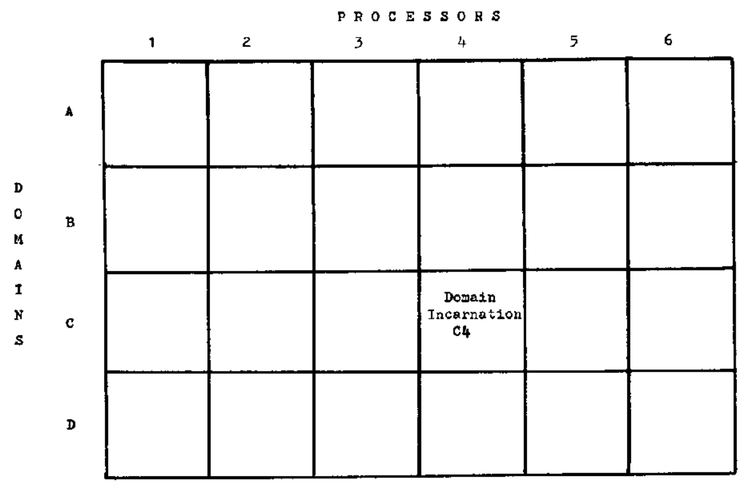


Figure 2: Domain Incarnations - the Intersections of Processors & Domains.

6.4 The Incarnation Own Permanent Segment which contains process-related local information which is preserved from one procedure invocation to another. It also is, by definition, unsharable among either domains or processes, hence mapped over as many physical copies as there are processes executing within that domain.

6.5 The Process Own Segment which is unsharable among processes, but shared among all the domains being "visited" by a single process's execution. It is mapped over as many physical copies as there are processes executing within the system, each process being exclusively associated with a single such copy. This copy is being "carried" along by the process on its interdomain journey (i.e., the process own segment will always be a window over the currently executing process's memory area), effectively enabling the meaningful exchange of arguments between domains.

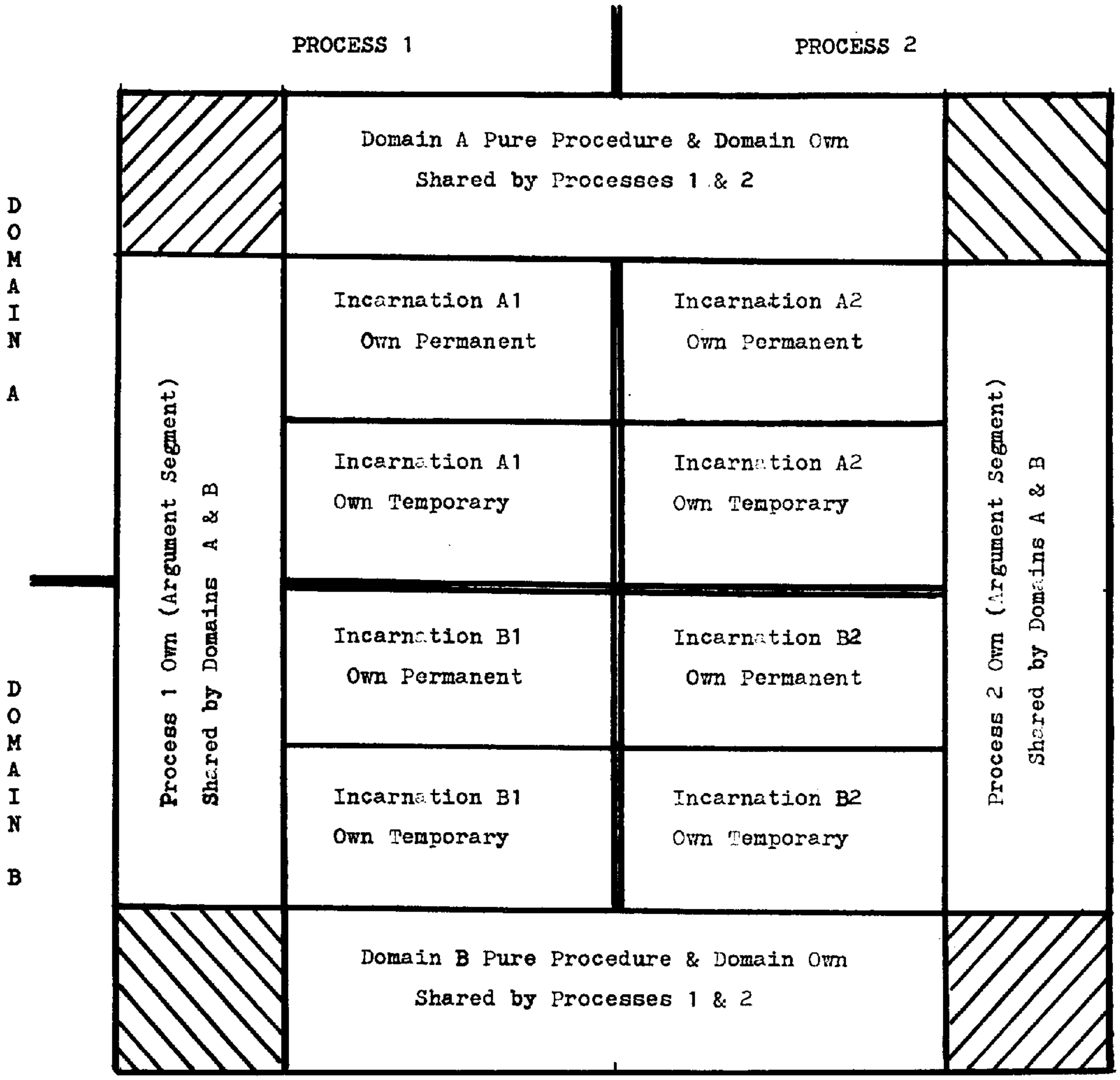
Figure 3 illustrates the intersection of two processes 1 and 2 with two domains A and B, and the way in which the various segments are (or are not) shared. We see that from an accessing point of view these segments may be classified into the following categories:

a. Segments whose single copy belongs to the domain and is shared among all processes, namely the procedure and the domain own segments.

b. A segment whose single physical copy belongs to the process and is shared by all domains, namely the process own segment.

c. Segments whose single copy belongs to the process incarnation (process/domain intersection) only, namely the incarnation own temporary and the incarnation own permanent segments.

Notice the distinction between memory space and address space. The above has just listed all the various memory residence requirements of the various segments, and the totality of all the physical copies mentioned forms the domain's memory space. From the "snapshot" point of view of a single domain being executed, it always consists of the five segments shown in Figure 1, being totally oblivious to the fact that these segments are in fact windows over different areas of memory. Thus, the proper implementation of a domain system essentially involves a generalized mechanism to always effect the proper mapping of the domain incarnation's current memory space.



PROCESS 1
PROCESS 2
DOMAIN A
DOMAIN B
Domain A Pure Procedure & Domain Own Shared by Processes 1 & 2
Incarnation A1 Own Permanent
Incarnation A2 Own Permanent
Incarnation A1 Own Temporary
Incarnation A2 Own Temporary
Incarnation B1 Own Permanent
Incarnation B2 Own Permanent
Incarnation B1 Own Temporary
Incarnation B2 Own Temporary
Process 1 Own (Argument Segment) Shared by Domains A & B
Process 2 Own (Argument Segment) Shared by Domains A & B
Domain B Pure Procedure & Domain Own Shared by Processes 1 & 2

Figure 3: The Sharing of Information between Processes and Domains.

We know of no programming language which supports semantics that would suitably designate the five storage classes just identified. M.I.T.'s project MAC pioneered the usage of a high level language -- PL/1 [C1] -- in programming the MULTICS system [O1] whose segmented, distributed-supervisor implementation displayed the need for essentially similiar storage-class semantics, though perhaps in a less obvious way. Their solution was: 1) to assign their own interpretation to the PL/1 storage-classes automatic, internal static, and external static [P3], and 2) make extensive use of explicit storage mapping through the device of PL/1 based structures[6].

We believe that the identification of these five storage classes is an important one, and that it might influence the design of higher-level languages destined to be used on future-generation domain-based operating systems.

[6]MULTICS was the first system where PL/1 based structures were actually employed as a major linguistic tool.

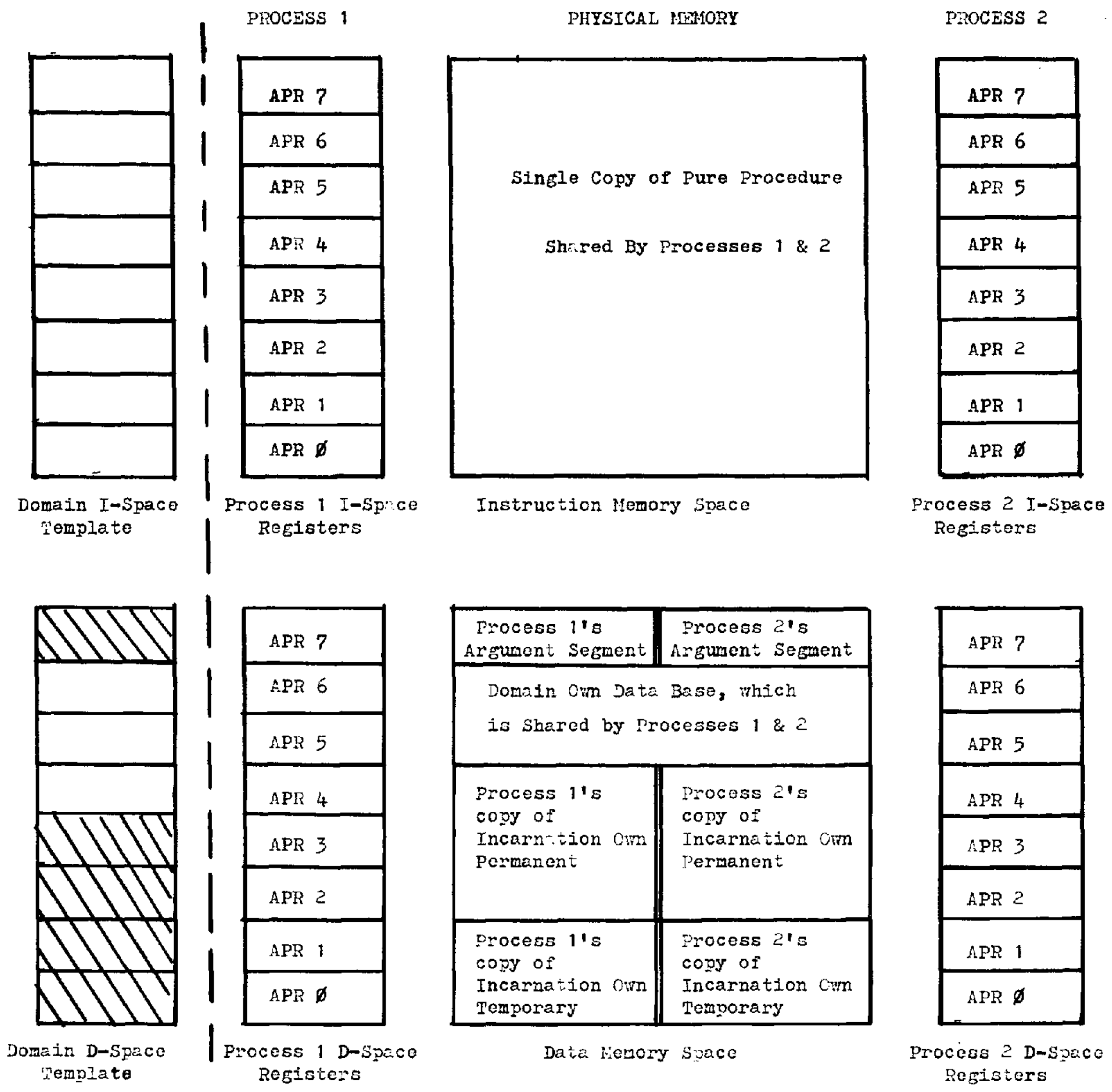

PROCESS 1
PHYSICAL MEMORY
PROCESS 2
APR 7
APR 6
APR 5
APR 4
APR 3
APR 2
APR 1
APR Ø
Single Copy of Pure Procedure
Shared By Processes 1 & 2
Domain I-Space Template
Process 1 I-Space Registers
Instruction Memory Space
Process 2 I-Space Registers
Process 1's Argument Segment
Process 2's Argument Segment
Domain Own Data Base, which is Shared by Processes 1 & 2
Process 1's copy of Incarnation Own Permanent
Process 2's copy of Incarnation Own Permanent
Process 1's copy of Incarnation Own Temporary
Process 2's copy of Incarnation Own Temporary
Domain D-Space Template
Process 1 D-Space Registers
Data Memory Space
Process 2 D-Space Registers

Figure 4: The Domain Mapping Technique used for our Actual Implementation. (note that I-Space & D-Space have each their dedicated set of 8 APR's).

7. THE ACTUAL IMPLEMENTATION

First, a word of explanation concerning the Memory Management Unit of the Digital Equipment Corporation's PDP-11/45 minicomputer [P2]. In each of the three protective modes (kernel, supervisor and user) all memory references are indirected through the memory space mapping Active Page Registers (APR's) where the effective address is re-translated into actual memory addresses. There are two sets of eight APR's, one set named Instruction Space (or I-space for short), the other Data Space (or D-space for short). All fetch-type references (i.e., references relative to the program counter register PC) are automatically directed to the eight I-space registers, all other references to the eight D-space registers. Each set of eight APR's provides a contiguous virtual address space of $(2^{**16}-1)$ bytes (=8 bits). The D-space may normally be read/written (each of the eight APR's has its own access control indicators), the I-space may only contain pure non-self modifying procedure and constants.

Each APR is a hardware register consisting of: 1) an address relocation base/boundary pair, and 2) a number of access control and programmed trap indicators. An APR may define a single contiguous area of storage ranging in size from 64 bytes to 8K bytes (K=1024), in discrete units of resolution of 64 bytes. Several APR's may map over a single area of storage to form nested, redundant or overlapping logical structures[7].

In our implementation, the domain's procedure segment was naturally mapped into the PDP-11/45's I-space, the remaining four domain segments were mapped into the D-space. The fact that the procedure's address space was somewhat out of proportion to the data segment's address space was considered immaterial in the sense that the I-space capacity provided an upper boundary, the actual procedure was typically much smaller, leaving a number of I-space APR's unused.

The mapping of the four data segments into the D-space was performed according to the following standard convention, as illustrated by Figure 4:

7.1 The process own segment was restricted to a maximum size of 8K bytes and always mapped into APR 7. Whenever a process crossed a domain boundary (through an inter-domain call/return) the entered domain's APR 7 was automatically remapped over the process's dedicated memory space.

7.2 The domain own segment was mapped into one or more APR's starting from APR 6 and extending through APR $\underline{m}$ (where $\underline{m} \leq 6$). This mapping remained constant for all domain incarnations.

7.3 The incarnation own permanent segment was mapped into one or more APR's starting with APR ($\underline{m}$-1) through APR $\underline{n}$ (where $\underline{n} \leq (\underline{m}-1)$). Whenever a process crossed a domain boundary, these APR's were mapped over an area of memory known exclusively to be associated with this domain incarnation.

7.4 The incarnation own temporary segment mapped into the remaining APR's ($\underline{n}$-1) through Ø. Its mapping was done in the same way as that of the incarnation own permanent segment.

For the reader who is curious about this top-to-bottom allocation scheme, a word of enlightenment. By nature of its CPU design, the PDP-11/45 stack mechanism causes the execution stack to progress in the order of decreasing effective addresses: thus if top-of-stack is address $1ØØØ_8$ pushing an item onto the stack will cause it to grow to address 777_8. Our top-to-bottom allocation scheme was designed to leave maximum expansion room for the stack.

8. THE KERNEL IMPLEMENTATION

The kernel contained only three major logical functions: 1) interrupt and I/O handling service routines, 2) CPU multiplexor and scheduler, and 3) memory management routines.

As mentioned earlier, in traditional systems the concepts of processor and memory space happened to overlap nicely, and were thus lumped together under the catch-all identity of user process. Thus, those systems typically had a single scheduler routine which would run a given user process by simultaneously setting the CPU runtime parameters and the memory space specifications (usually a base/boundary register). In our system, these three functions were kept separate. The CPU manager would only govern the current duration of time allowed to any given user process. The memory space switching function was an independent one, invoked by the (kernel software implemented) inter-domain call/return mechanism. Each process was defined by a dedicated data base known as the Process Activation Map (PAM), and the CPU manager's table entry consisted of only two logical items: 1) scheduling information such as run-time quantum, scheduling priority, and running-status indicators, and 2) a pointer to the process's PAM. This organization allowed us to make the PAM into a swappable data base, independent of the core-locked CPU manager's table.

[7]Compare this to the descriptors proposed in Spier's model mechanism [S3]. We could, unfortunately, make no extensive use of this desirable feature because of the obviously limited number of APR's, although the feature did come in handy on certain occasions, where complex mapping was called for.

The PAM contained two logical tables: 1) an inter-domain return stack, which was a software simulation of the model hardware mechanism's [S3] return stack register, and 2) a known domain table (KDT) containing APR images for all the domain incarnations currently known to the process. A current domain pointer variable always pointed to the KDT entry corresponding to the process's current domain of execution.

Thus, whenever a process was scheduled to run, the CPU manager would simply reload the hardware APR's with the appropriate image from the KDT as designated by the current domain pointer. An inter domain call/return essentially consisted of 1) proper handling of the inter-domain stack[8], and 2) the repositioning of the current domain pointer to the appropriate KDT entry, following which the APR's would be reloaded from the new KDT entry.

Notice that the kernel itself was only a domain among many, albeit the most privileged one. Its code resided in the procedure segment. Systemwide data bases (such as the CPU manager's table) resided in the domain own segment, and the PAM and the execution stack in the corresponding two incarnation own segments. Being the guy in charge of memory mapping, the kernel had to be aware of the domain incarnation concept and whenever it switched its services from one process to another, it explicitly had to re-map its domain incarnation segments. To allow the kernel to execute during process-independent transitional states, it had an additional small stack area in the domain own segment; however, for most of its code it executed as a true kernel-incarnation, a fact which greatly simplified the function switching among processes, even allowed the kernel to call out to another supervisory domain. Namely, while the kernel would execute on a systemwide basis (using the domain own segment stack) it was logically uninterruptible. When executing as a true domain incarnation it could be logically interrupted in the dual sense of process switching or calling out to some supervisory domain. The kernel-incarnation's interrupted state was saved in the appropriate incarnation own memory areas, protected and inviolable until resumed at some future time.

Thus, the kernel whose purpose it was to manage domains, was itself an auto-managed domain, differing from all others by the following technical points: 1) it had to do its own memory mapping, and 2) it needed an additional small single stack for the times when it executed as a process-independent entity.

9. THE MAKING OF A DOMAIN

Until now we have discussed the implementation of a domain in terms of already existing domain incarnation memory space maps, duly represented in the form of APR images stored in individual processes' KDT's. We shall now briefly discuss how those memory space maps came into existence.

For our program coding we used the PDP-11 MACRO-11 assembler. The proper allocation of bodies of code onto their logical storage class had to be done explicitly, through judicious manipulation of the assembler's location counter. The output of the assembler consisted of two distinct core images, one for the I-space and the other for the D-space. An additional output item consisted of control information identifying the five domain segments' relative addresses.

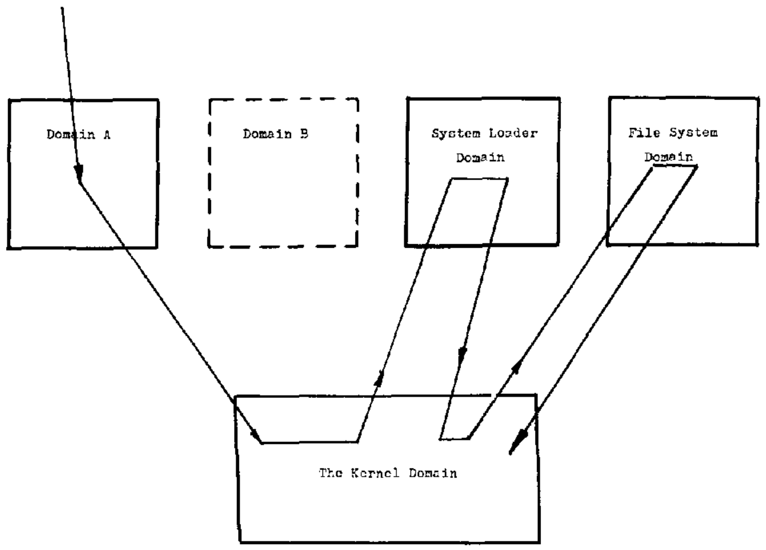


Figure 5: Typical Flow of Control in Making Domain B Known.

This assembler output now resided in the system's secondary storage under the file identity NAME.OBJ, known as the domain's object file. The system's disk was subdivided into two logical entities: 1) the file repository, and 2) the run-time swapping area. When a domain was called for the first time in the system, its object file would be copied onto the swapping area, and a pair of APR template images prepared (see template illustration in Figure 4). When a process called a domain which was unknown to that process (i.e., did not have a KDT entry), the system would generate in the swapping area copies of the incarnation own information, generate special domain incarnation APR images and store them in the process KDT. Compare this technique with TENEX's copy-on-write facility [B2].

This preparation of domain incarnations on the system's disk swapping area was not a function of the kernel, but rather of the system loader residing in its own dedicated domain. And while the kernel had to be

[8]Our kernel implemented call/return mechanism was effectively more sophisticated, providing uniform handling for condition signals, abnormal returns and inter-process communication, all of which would result in a standard call to a specified procedure entry point, to be uniformly dismissed via a standard return. This comprehensive mechanism is described in a companion paper [S6].

locked in core, the system loader was swappable. When the kernel's call mechanism detected an attempt by some process to call an unknown domain, the kernel simply performed an inter-domain call to the system loader. Upon return from the loader, the kernel would carry through the interrupted call sequence. This ability of the system's most privileged component to call out to some lesser privileged module is a most important characteristic of the domain architecture. Contrast this with the typical user/monitor arrangement (or even with ring structured systems) where the inter-domain call hierarchy is strictly unilateral, always going towards the higher-privileged domain.

Figure 5 illustrates a typical flow of control, where some process executing in domain A attempts to call domain B which currently is unknown. The kernel interrupts the attempted call to invoke the system loader, which in turn may invoke the file system, which in turn may call on the kernel to perform some disk I/O in the file system's behalf.

10. FUNCTIONAL IMPLICATIONS

Our actual implementation did not go beyond the limits described up to now. Which is not to say that our thought processes have necessarily ground to a halt. Even though we felt that any further active implementation would surpass the research goals that we had set, we did engage in additional conceptual design so as to gain further insight into the functional implications of this architecture.

For example, it seemed clear that by separating the traditional file system into two distinct logical modules, 1) disk space manager which would manipulate disk files, each known by its systemwide internal unique name, and 2) a name manager which would associate a symbolic (possibly hierarchical) nomenclature with internal unique file names, we could support a multitude of differently flavored file systems, simply by having several name manager domains. Similiarly, by supporting several login manager domains, we could effectively run several disparate operating systems on the same hardware base.

To effectively demonstrate this architecture's simplifying influence on traditionally complex functions, we shall present our design for a surprisingly simple-minded, yet extremely flexible CPU scheduler.

In an interactive operating system, a user process is typically in one of three states, each requiring its own special scheduling parameter, namely: 1) highly interactive (e.g., editing) where the system response must satisfy certain human engineering requirements, such as guarantee response within a certain maximum delay of 1-3 seconds, 2) compute bound (e.g., heavy handed compilation) where the user knows that response will not be immediate, thus his process may be put into some lower priority queue, and 3) real time (e.g., interrupt handling, file I/O) where the process is temporarily granted the very highest scheduling priority.

Current system schedulers normally recognize the fact that a process has switched into real-time mode, but otherwise are unable to exactly know what scheduling parameters are needed. They therefore consist of highly complex statistical algorithms to make educated guesses about the process's current scheduling needs, based on the process's past behavior. In our projected design, the scheduling algorithm would consist of simply enforcing the current scheduling parameter, as posted in the process's scheduler table entry. The value of the parameter would come from the executing program itself. It is the only entity in the system that knows for sure what its scheduling requirements are. Whenever these requirements change, it would call the scheduler and post the new parameter value in the process's table entry. Thus, when the editor is entered, it would set a parameter corresponding to highly interactive activity, the compiler a parameter corresponding to prolonged computation, the disk space manager a real-time parameter.

The only reason why such simple-minded scheduling is not implemented on the traditional system relates to system security. Given the ability to influence system scheduling through explicit monitor calls, the malicious user could completely dominate the entire system resources to the detriment of the other users. If we recognize, however, that a major part of the programs executed during a typical user session are in fact system-provided, and thus certifiable, and that in a domain system there is no way in which one can modify the behavior of a protected system facility, we realize that there would be no protection loophole in allowing system library modules to set their own scheduling parameters. User coded modules would be known as such, and assigned default parameters no matter what they request. A user who sincerely requires specialized scheduling would have to relinquish his program to the system administrator who (following suitable certification) would grant the request, after having however removed that module from the user's access range, and thus made it immune to further modification by the user.

We envisioned two scheduler modules, 1) the kernel resident micro scheduler which enforces parameters posted in the scheduler table, and 2) the external supervisory-domain resident macro scheduler which would be able to modify the various scheduling parameter values associated with the various domains. By properly manipulating the macro scheduler, the system administrator would be able to flexibly tune the overall system throughput performance. Compare this to existing schedulers, where any kind of tuning may call for the bothersome recoding of parts of the monitor, which in turn may provoke all kinds of unrelated bugs.

11. SOME AFTERTHOUGHTS

The most startling effect that this experience has had on us was the way in which this architecture forced our thought processes towards modularity and concise functional identification. As mentioned, we could not just go ahead and implement a traditional user process, we had to delve into the functional distinctions between memory space, address space and processor. We could not just implement a traditional CPU multiplexing procedure, we had to distinguish between the CPU manager and the memory space manager, the micro-scheduler and the macro-scheduler.

Thus, in a curious feedback process, the mechanism which was originally designed to facilitate modularly structured programming, effectively forced our hands and insisted that it itself be properly modular. And while this is not to say that a domain-based system could not be sloppily implemented, there is hope that by merely adopting this architectural approach operating systems would somehow tend to become cleaner, more comprehensible and easier to maintain and modify. In our opinion, the geodesic operating system is within the realm of the possible, contingent only upon the emergence of next-generation domain oriented hardware machines.

REFERENCES

B1) Barton R S, "A Critical View of the State of the Programming Art", Proc. 1963 SJCC, pp. 169-177.

B2) Bobrow G D, "TENEX, A Paged Time Sharing System for the PDP-10", CACM March 1972, pp. 135-143.

C1) Corbato F J, "PL/1 as a Tool for System Programming", Datamation vol 15 #6 1969, pp. 68-76.

D1) Denning P J, "The Working Set Model for Program Behavior", CACM May 1968, pp. 323-333.

D2) Dennis J B, "Segmentation and the Design of Multiprogrammed Computer Systems", ACM Journal vol 12 #4, pp. 589-602.

D3) Dennis J B, Van Horn E C, "Programming Semantics for Multiprogrammed Computations", CACM March 1966, pp. 143-155.

D4) Dennis J B, "A Position Paper on Computing and Communications", Proc. 1st ACM SIGOPS SOSP, October 1967.

D5) Dijkstra E W, "The Structure of the 'THE' Multiprogramming System", CACM May 1968, pp. 341-346.

D6) Dijkstra E W, "Complexity Controlled by Hierarchical Ordering of Function and Variability", Proc. 1968 NATO Conference (N1), pp. 181-185.

D7) Dijkstra E W, "Hierarchical Ordering of Sequential Processes", Acta Informatica vol 1 #2 1971, pp. 115-138.

E1) Elspas B, et al., "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys, vol 4 #4 1972, pp. 97-147.

E2) Evans D C, Leclerc J Y, "Address Mapping and the Control of Access in an Interactive Computer", Proc. 1967 SJCC, pp. 143-155.

F1) Fano R M, "The Computer Utility and the Community", IEEE International Convention Record, Part 12 1967, pp. 30-37.

F2) Feustel E A, "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, vol C-22 #7, July 1973, pp. 644-656.

G1) Graham R M, "Protection in an Information Processing Utility", CACM May 1968, pp. 365-369.

G2) Graham G S, Denning P J, "Protection - Principles and Practice", Proc. 1972 SJCC, pp. 417-429.

H1) Habermann A N, "Prevention of System Deadlocks", CACM July 1969, pp. 373-377.

H2) Hoffman L J, "The Formulary Model for Flexible Privacy and Access Control", Proc. 1971 FJCC, pp. 587-601.

J1) Johnston J B, "Structure of Multiple Activity Algorithms", Proc. 2nd ACM SOSP, October 1969, pp. 80-82.

L1) Lampson B W, "Protection", Proc. 5th Princeton Conference on Information Sciences and Systems, March 1971, pp. 437-443.

N1) 1968 NATO Conference Report, Naur & Randell Eds, NATO Scientific Affairs Division, Brussels 39, Belgium.

N2) 1969 NATO Conference Report, Buxton & Randell Eds, NATO Scientific Affairs Division, Brussels 39, Belgium.

O1) Organick E I, "The Multics System: An Examination of its Structure", M.I.T. Press, Cambridge MA, 1972.

P1) Parnas D L, "On the Criteria to be Used in Decomposing Systems into Modules", CACM December 1972, pp. 1053-1058.

P2) PDP-11/45 Processor Handbook, Digital Equipment Corporation, Maynard MA, 1971.

P3) PL/1 Language Specifications, ECMA.TC10/ANSIX3J1, PL/1 Basic/1, December 1972.

S1) Schroeder M D, Saltzer J H, "A Hardware Architecture for Implementing Protection Rings". CACM March 1972, pp. 157-170.

S2) Schroeder M D, "Cooperation of Mutually Suspicious Subsystems in a Computer Utility", M.I.T. PhD Dissertation, Proj. MAC TR-104, September 1972.

S3) Spier M J, "A Model Implementation for Protective Domains", International Journal of Computer & Information Sciences, vol 2 #3, 1973.

S4) Spier M J, "A Pragmatic Proposal for the Improvement of Program Modularity and Correctness", Submitted for Publication, International Journal of Computer & Information Sciences.

S5) Spier M J, "Process Communication Prerequisites, or the IPC-Setup Revisited", Proc. 1973 Sagamore Conference on Parallel Processing, IEEE Catalog #73 CH0812-8C, August 1973.

S6) Spier M J, "The Experimental Implementation of a Comprehensive Inter-module Communication Facility", Proc. 1973 Sagamore Conference on Parallel Processing, IEEE Catalog #73 CH0812-8C, August 1973.

S7) Spooner C R, "A Software Architecture for the 70's: Part I - The General Approach", Software Practice & Experience, vol 1 #1 1971, pp. 5-37.

V1) Vanderbilt D H, "Controlled Information Sharing in a Computer Utility", M.I.T. PhD Dissertation, Project MAC TR-67, October 1969.

W1) Wegner P, "Structured Programming, Program Synthesis and Semantic Definition", Center for Computer & Information Sciences, Brown University, TR-72-63, September 1972.

W2) Wulf W S, et al, "HYDRA: A Kernel Operating System for C.mmp; External Specifications", Department of Computer Science, Carnegie Mellon University, October 1971, internal preliminary notes, unpublished.