Overview of the Program and Command Environment With the Executable Program Format (EPF) And Virtual Memory File Access (VMFA)

David Udin

Table of Contents

Overview of the Program and Command Environment With the Executable Program Format (EPF) And Virtual Memory File Access (VMFA)

David Udin

1 Introduction

The introduction of the executable program format and virtual memory file access allow us to significantly strengthen principles of software architecture that previous developments have been directing us towards. The advances are in the ability to use programs in a subroutine-like manner, in the ease of preparing collections of procedures for binding at run-time, and in the ability to use the file system as an extension of virtual memory.

I assume the reader of this document has some familiarity with the current program environment and with the overall nature of Prime V-mode architecture.

2 Executable Program Format

The Executable Program Format is a new format for a run-file, the object that represents an executable program, analogous to the V-mode SEG run-file or the R-mode saved file. Unlike the SEG run-file or the R-mode saved file, which represents a program as a memory image of both procedure and data, an EPF represents a program as an image of the procedure portion, and a description of the data portion (link frames and COMMON) to be constructed before running the program.

(I use EPF as an abbreviation both of Executable Program Format and of Executable Program Format file, that is, a file in this format; it should be clear from the context which use is intended.)

The significant features of the new format and of its treatment by PRIMOS are:

- An EPF can be directly mapped into virtual memory (by VMFA) without modification.
- A segment of an EPF can be assigned any address at run-time. The linkage described by an EPF can be constructed (almost) anywhere. Consequently, EPF's can coexist with each other in memory and be shared among processes without permanent reservation of addresses or installation in public address space.
- All of the information about a program is in its EPF and can be made available in memory at run-time.

- The command environment will map into memory and initialize for execution programs and commands constructed as EPF's: there is no need to invoke a command to transform and invoke and EPF.

- Run-time libraries can be built and installed in the format. Libraries especially benefit from the location-independence of the format. A library built as an EPF automatically includes a hash table of its entry points for searching by the dynamic linking machinery.

2.1 Format

The principal components of an EPF file are the procedure segment images and the description of the linkage areas to be constructed when the EPF is invoked. Other components are the entry point table, load map, and information for the source level debugger.

The EPF is contained in a DAM file, formatted as a sequence of segment images that can be mapped directly into virtual memory by VMFA. The segments of the EPF need not be assigned consecutive addresses in memory; each is separately relocated according to the availability of address space at run-time. All but the last of the EPF's segment images must be a full 64K in length; that is, segments begin on 64K boundaries in the file, a constraint imposed by VMFA (see section 4).

The first segment of the EPF contains a few words of information about the size and structure of the EPF and the program it represents, and a pointer to the structures containing the descriptive portion of the EPF. The first EPF segment is also the first procedure segment image. If the EPF includes more than one procedure segment they are the segments immediately following the first. Immediately following the procedures in the last procedure segment image is the beginning of the descriptive material: linkage descriptors, entry point table, load map, and debugging information. Thus, if the program is a short one, a single segment might serve to hold all of the EPF.

There are no references within an EPF by absolute addresses; typically, there are no absolute addresses in an EPF at all. All inter-segment references within the EPF and all references outside the EPF are through indirect pointers, residing in link frames, which will be filled in at run-time. Inter-segment references within the EPF will be relocated when the EPF linkage is being constructed when the program is invoked; references external to the EPF (which have their fault bit set) will be relocated when the fault is encountered and the address determined and filled in by the PRIMOS dynamic linking mechanism. Thus, the format allows the operating system to assign a procedure segment image to any available segment address.

Link frames and COMMON areas are collected into linkage areas which are relocated as a unit. Grouping this material into contiguous storage allows sharing of faulted external references to reduce the overhead of dynamic linking. Usually a linkage area will be less than a segment long. Multi-segment linkage areas are used only where at least one of the objects in it is longer than a segment and thus requires long

addressing anyway. A linkage area less than a segment in length can be constructed at run-time in any region that lies within a segment. This word relocatability is provided because, for a small increment of cost over the cost of segment relocatability (which is necessary anyway), it permits combining linkage for several EPF's into a single temporary segment at run-time. A minor constraint on the relocatability of multi-segment linkage areas is that they must be aligned with segment boundaries.

Because there is no modification of the procedure segment images to relocate them, and because the linkage is constructed in storage allocated at run-time and not in the EPF itself, the EPF can be read-only and shared among processes as well as mapped into different addresses in each process using it, all without special effort on the part of the system designer. Furthermore, one invocation of an EPF can be suspended and the EPF invoked again in the same process with a different copy of the linkage areas without interfering with the suspended invocation. This means that commands built as EPF's can behave like internal commands, interacting only in well-defined (presumably intended) ways, rather than through their accidental interference in memory. I will discuss this further in the section on the command environment.

Library EPF's contain a hash table of names of external entry points; program EPF's contain a main entry point. An EPF can have both and be used interchangeably or, indeed, simultaneously as both a main program and a collection of subroutines to be linked to at run-time. The distinction is contextual: the main entry is used if the EPF is invoked as a command; the other entry points are used to satisfy pointer faults to "dynamic entries". More on this distinction in the section on the command environment.

As with the SEG run-file, all the information about the program needed for any purpose at run-time is in the EPF; however, unlike the SEG run-file, all the information in an EPF can be referenced in virtual memory, relative to the actual addresses assigned the EPF, once the EPF has been mapped into memory. (It may be necessary to request completion of the mapping of a large EPF where some of the segments, e.g. those containing the debugging information or the load map, have been left out of virtual memory until required.)

2.2 BIND

BIND is the command for producing EPF's from compiler object text. Because the nature of EPF's and their treatment by PRIMOS is such that one rarely specifies where programs are to be loaded, BIND's user interface can be much simpler than SEG's. We expect that most of the time all that will need to be specified to BIND is the list of object files that are to be made into an EPF and the name of the result, and those can be specified in the command line to BIND. There is also a command mode to BIND for specifying additional information, such as the names of entry points when building a library EPF.

2.3 Relation to VMFA

EPF's do not depend on VMFA for their existence or use. VMFA does provide direct inclusion of an EPF in address space, sharing (simultaneous use of an EPF by more than one process), and support for the larger address space that might be necessary for extensive use of activities that EPF's make possible: nested invocation of external commands and use of per-process dynamically-linked libraries. Features provided by full VMFA could be provided by other means and, in fact, will initially be provided by a very restricted version of VMFA (read accesss to existing files only).

2.4 Future Development

One very important feature missing from the initial implementation of EPF's is a representation of dynamically linked or dynamically created data areas. With this feature it would be possible to specify deferring creation of data areas (e.g. Fortran COMMON areas) until reference, and to specify linking at run-time to data external to the EPF. (Initially, it can be done in somewhat restricted ways by subterfuge.) Support for this feature in EPF's has been designed and will eventually be included. Support for this feature in the command environment has not been designed: the question is how to specify in a sensible, natural way the association of internal names with external objects (usually files).

There are two reasons for desiring run-time linking to data. First, subroutines that refer to the same COMMON areas can be bound into different EPF's. Second, mapping of language objects to file system objects can be performed without explicitly including in programs operating system requests that are sensitive to machine architecture and calling sequence.

3 Programs, Libraries, and the Command Environment

What are the ingredients the command environment must contend with? An EPF represents part or all of an uninitialized program or run-time library. A program is not confined to a single EPF; it can link at run-time to, and thus include, subroutines from other EPF's. Before the EPF can be executed the operating system must assign it to addresses and construct the linkage regions that go with it. Using an EPF doesn't necessarily wipe out the previous one the way using today's run-files usually does; both the representation of the uninitialized program or library (the EPF) and the initialized linkage material from a previous invocation can linger in address space. Thus the operating system must discover program boundaries, determine when to initialize (or re-initialize) an EPF, and decide when it is necessary or desirable to discard used linkage areas.

3.1 Programs and Static Storage

What is the extent of a program? It is a main program and all the subroutines it calls and all the subroutines they call and so on; "transitive closure under procedure call" for you mathematicians, "program baggy" for those of you who like catchy names. The lifetime of an invocation of a program is from the call of the main program to its return.

Associated with a program is a certain amount of static storage: COMMON or external static and link frames with their static variables, entry control blocks, indirect pointers, etc. Some of the static variables may have been specified to have an initial value. All of the static storage must persist until the program is finished running. Thus, sometime before a subroutine in the "program baggy" is entered for the first time its static storage must be allocated and initialized. That storage must persist for the lifetime of the program, which is defined to be until the main program returns. If the main program returns and is invoked again, the static storage for any subroutine in its baggy must be re-initialized before the subroutine is entered in this subsequent invocation. Whether the storage from the previously completed invocation was reclaimed and new storage allocated and initialized for the second, or whether the storage from the first was retained and simply re-initialized in between program invocations is irrelevant to the correctness of the treatment of static storage; however, it may have important consequences for the performance of the system, as I will discuss later.

Up until now I have used the term "invocation" somewhat loosely. Let us define "invoke" to be what you do to a program, as opposed to calling a subroutine. That the actual transfer of control to a program may be implemented with a procedure call instruction in the machine does not affect the distinction; the distinction is based on the treatment of static storage: invocation requires that static storage be placed in its initial state; subroutine call does not.

(Note that MULTICS does not make this kind of distinction between invocation and procedure call. In MULTICS there is only the concept of the procedure call, and thus, by our definition of program, there is only one program per process. Static storage for any subroutine in that program/process is allocated and initialized only once and persists until the subroutine is explicitly removed or the process is explicitly reset to its initial state. We choose to make the distinction because it corresponds to the distinction between programs and subroutines in today's environment (MULTICS had to incorporate a special mechanism to run FORTRAN programs), and because it results in more predictable behavior of nested program invocations. More on this in the discussion of command levels.)

A subroutine linked to at run-time (that is, a subroutine in a library EPF) is functionally no different from a subroutine incorporated by BIND into an EPF with the main program: before a subroutine in a library can be called as part of a program an initialized copy of the static storage it uses must be constructed. If the subroutine is called by a different program invocation during the lifetime of the first one (as might happen if the first program is suspended by a QUIT, and the same program or another invoked) then it must be called with a different initialized link frame. The only difference between subroutines in the same EPF and subroutines in different EPF's but called as part of a single program invocation is the time of their initialization: all the linkage material for an EPF will be constructed and initialized at the time of first linking to some subroutine in it. Subroutines linked to subsequently from the same invocation will use that same linkage. In other words, the EPF is the unit of initialization.

3.2 When to Initialize

What indicates the boundaries of a program invocation to the command environment? Program invocations are defined to be distinct if executed at different command levels or if executed serially in time within a level. In other words, we require that to suspend a program or to otherwise perform a nested invocation of a program you must move up a level in the command environment. There must be an explicit call to create a new level, an error signal, or a QUIT in order to indicate to the command environment that the program under execution is to be suspended, not discarded. A suspended program invocation is retained until control returns to that level (by returning or proceeding from the level above) or until the entire level is released. The other side of requiring the user (or a program) to explicitly tell the environment to not discard a program invocation is the assumption that if a user invokes a command at some level he is finished with the command previously invoked at that level and the command environment is free to discard it. Note that by waiting until the next command is uttered at a level to discard the previous program invocation, rather than discarding it immediately on return from the program, we give the user a chance to obtain a new level (by a QUIT, for example) from which to perform a post-mortem on the storage of a completed program.

How does the operating system tell when to allocate and initialize static storage for a particular EPF, which may or may not already be mapped into address space and may or may not already have one or more copies of its static storage hanging around? One way the operating system could do this is to discard all the linkage material previously allocated by a command level every time a new program is invoked at that level. Then when a fault in a pointer to an external reference is being processed (called "snapping a link") software need only check to see if the library EPF containing the entry point that satisfies the faulted link has an initialized copy of its static storage at the current command level. If it has, use it. If it hasn't, allocate and initialize storage, and remember somewhere that it has a copy of static storage for that library at that level. The drawback of this approach is that the command processor keeps reclaiming and re-allocating storage for a library in frequent use.

Instead, each command level maintains a "program sequence counter", and each time the command level invokes a program it increments it. When a library EPF is linked to for the first time at a level and static storage allocated and initialized, the counter is saved in a data structure associated with that level's use of the library. Thereafter, whenever a pointer fault at that command level leads to that library the saved program sequence counter is compared with the level's current program sequence counter; if it is different it means the fault comes from a new program invocation, and the library is re-initialized and the saved counter updated; if it is the same it means the fault comes from the same program invocation as the library's static storage, and the link is made without initialization, and the saved counter is left unchanged. In other words, the saved counter serves to identify which program invocation the current version of static storage for an EPF "belongs to". If we have advanced to a new program, the static storage is out of date and needs re-initialization. Otherwise, it is still current and should be left alone. A separate program sequence counter and list of initialized libraries is maintained at each level so that use of libraries at different levels, which we know must be by different program invocations, is independent, that is, uses different copies of static storage.

The overall result is that for a given EPF different levels will allocate different storage and thus allow suspension of a program invocation and simultaneous use of part or all of its constituent EPF's at a higher level without interfering with the suspended invocation. For a given library EPF used by different program invocations at the same level, the level will simply re-initialize storage it has previously allocated.

In the first implementation we are going to assume that programs are not re-used sufficiently frequently to warrant retaining their storage the way we do for run-time libraries; the command level will simply discard storage for an EPF invoked as a main program as soon as the next program invocation is made at that level. It is a trivial matter to change the implementation to allow storage for main program EPF's to linger in address space and be re-initialized and used at subsequent invocations of the same program if we find that there is anything to be

gained by it. Furthermore, one can get the same effect by installing most of a commonly used program as a library with a very short program EPF serving only to call the library version, much as the shared editor is treated today.

3.5 What to Initialize

Re-initialization of static storage used by a previous invocation of a program must include resetting faulted indirect pointers, as well as giving variables their requested initial values. This is because the fault is the only way that the operating system has to detect that a library is about to be entered as part of a program invocation. For example, if program P calls library A a fault occurs when a reference is made to the indirect pointer (IP) referenced by the call. The dynamic linking software finds the location of the entry to the library, initializes the EPF containing it if necessary, and places the address in the IP and resets the fault bit. If the routine in library A calls some routine in library B there will also be a fault, causing that library to be prepared and the link to it from A to be snapped. At this point if P were to call some routine in B the link would be snapped, but B would not be initialized, because it has already been initialized as part of P's program invocation; that is, B's saved program sequence counter matches the level's current value. Now suppose P completes and another program, Q, is invoked which calls A, causing A to be re-initialized because Q is a new program invocation, reflected in a program sequence counter greater than the one saved with the last initialization of A. If this initialization of A does not reset the faulted IP's ("unsnap the links") in A's linkage, a reference through a previously snapped link to B will not cause a fault and the operating system will not notice that the static storage for B is out of date and in need of initialization. (Worse still, if A called both this snapped link to B and some other, as yet unsnapped, link to B, it would cause an initialization at an inappropriate time.)

3.4 Categories of Libraries

Suppose a library doesn't have any static working storage; why keep on re-initializaing it? For a library with no static storage we allow designating its EPF as a "process-class" library. A library so designated will only be allocated storage for linkage and initialized once in the lifetime of a process (and thus the name "process-class"). Subsequent dynamic links to such a library will not cause re-initialization, regardless of the level or program invocation from which they come. Since outward dynamic links from such a library (which are in static storage, of course) will also never be re-initialized, we don't allow links from a process-class library to ordinary libraries ("program-class" libraries), because it would interfere with the detection of program boundaries. For example, suppose a program invocation links to a process-class library, causing allocation and initialization of its static storage. The process-class library then in turn links to a program-class library during that first invocation. When the first invocation returns, another program is invoked and happens to link to the same process-class library the first one linked to. But this time the process-class library is not

re-initialized, so if it calls the same program-class library it did the last time there will not be a pointer fault, and the program-class library will not be re-initialized even though it is being used as part of a different program invocation. Furthermore, if the level of the program-class library were released, giving up all its static storage, the pointers to any entries in it from the process-class library would be left dangling.

Since a subroutine in a process-class library can be suspended by a QUIT and later entered from a higher command level it is safest to not have any static working storage in such libraries. But it is not required that that be the case: one could write a library subroutine that counted the number of times is was called, for example. (But note that you would have to be careful to suspend QUIT's or use a store-conditional instruction at the critical point of reading and updating the counter.)

There is also a "level-class" of library. Static storage for such a library is allocated and initialized only once at a given command level. The consequence is that a level-class library can use static storage, and the storage will not be interfered with by use of the library at other command levels, but the storage will not be re-initialized between program invocations within the level. Thus the static storage of such libraries can be used to transmit information between serial program invocations within a level. Another use for this class of treatment of static storage is for a library which uses a significant amount of static storage but which initializes the storage itself, rather than depending on the operating system initializing its storage. An editor in this form could: 1) be used as a subroutine, 2) be used at any level, even if it had been quit from at a suspended level, and 3) be re-initialized more efficiently by code internal to it rather than be re-initialized by the pointer fault software following the description in the EPF. As with process-class libraries, we don't allow linking from level-class to program-class libraries because it interferes with the detection of program boundaries by link faults and because it would lead to dangling pointers if the command level were released. We do allow linking to process-class libraries from level-class libraries, however.

3.5 Classifying the Effects of a Program

It should be clear by now that the treatment of storage associated with an EPF and the location-independence of an EPF strengthen the role of the command level: a significant result of the introduction of EPF's and the associated changes to command processing (VFMA, too - more on this later) is that it becomes straightforward to make use of whole programs as readily as using a subroutine. For example, any interactive subsystem that is an EPF and follows the "recursive" rules could have a subsystem command for executing a PRIMOS command without "leaving" the subsystem environment with a QUIT or similar escape. While programs should no longer interfere with each other by the accident of their location in memory, there are still many spheres in which they can legitimately interact. I think it is helpful to categorize those areas of interaction in a hierarchical manner.

Outermost is the system environment, containing everything shared among processes: file system, semaphores, networks, virtual memory in DTAR0 and DTAR1, service processes like the line printer spooler, and so forth. We could further subdivide this environment, perhaps, but it wouldn't make much difference from the point of view of a process running on a machine, which is our principal concern here.

Next in the hierarchy is the process environment, containing all that is global to a process: the file units, the private portion of virtual address space (DTAR1 and DTAR2), the terminal, assigned devices, global command variables, home and current UFD's, abbreviations, semaphores, static storage of process-class libraries, and so forth.

Next in the hierarchy is the level environment, any number of which may coexist simultaneously, the topmost of which is active: the one the user is talking to when he is issuing commands. Programs invoked at different levels should interact only through their effect on the process environment. At a given level, a program invocation may affect subsequent ones by its effect on the level environment, but at this stage of PRIMOS development there is not much in the level environment for programs to interact through. (It includes a QUIT inhibit counter and static storage for level-class libraries.) In essence, the current emphasis is on using levels to isolate a suspended program invocation; in the future we might also wish to put some emphasis on serial continuity of programs within a level, such as by including some kind of stream notion for communicating the output of one command to the input of the next.

Innermost of these environments is the program environment, of which there is one per level at any given time. These interact only through their effect on the process environment and through the environment of the level within which they are executed. programs that are executed as different levels can only affect each other through the process environment; programs at the same level can interact through their effects on both the process environment and the level environment. The program environment consists of static storage for its single program invocation.

The kind of independence we are talking about here is conceptual. Since all of these environments coexist in virtual memory in the same machine protection ring (ring three) they are not prevented from accidentally mutilating each other, but if the programs obey the rules one can suspend one program and execute another and then return to the suspended program without interfering with it in unforeseen ways. In other words, a program should have some "official" effects on the system, process, perhaps the level, and whatever it communicates directly to the user. Those effects are the purpose of the program; anything else should be considered as unwanted side-effect. To be able to say that a program has "no effect" on some part of the environment is not to say that it does nothing to the environment, only to say that it follows rules such that it returns the environment to its initial state. For example, if a program uses dynamic assignment of file units and closes any unit it opens, one can say that it has no net effect on

the file units. If it operates on some specific unit, however, those operations are part of the prescribed effects of running the program: if you don't describe those effects as part of the purpose of the program, they can only be considered as unwanted "side effects": a bug. Similarly, temporary storage should be obtained from and returned to the operating system explicitly: getting segments by simply referencing specific addresses plays havoc with the nested view of levels.

3.6 Treatment of EPF Libraries by the Command Environment

The new executable program format incorporates the information necessary for run-time linking to a subroutine in a library EPF and removes the need for fixed address assignments for a shared library and its associated linkage. The BINDer will produce a library EPF without the need for sophisticated knowledge on the part of the library's creator. VMFA makes it possible to share without the use of public memory and use an EPF without making a copy of it. These new capabilities greatly lower the threshold of sophistication and administrative nuisance required to employ PRIMOS's version of dynamic linking; we are also introducing a modest increase in the flexibility of specifying the search order of run-time libraries.

In addition to supporting the current library mechanism, we will introduce a system-wide library search list, which serves the same role as the current search mechanism but for EPF libraries, and we will introduce a per-process library search list for the individual specification of libraries to be searched. The per-process search list makes possible the use of run-time linking without full public access to the set of routines; access to a library for inclusion in a per-process search list is controlled by the normal file system access control mechanisms. It also makes it possible for a process to substitute its own version of a library for a system-wide library. These new features should have a considerable impact on use of "dynamic binding" in our systems.

It is not intended that this method is all of what some people mean when they talk about "dynamic binding" and "search lists". In particular it differs from MULTICS in that it searches specified libraries, not specified directories. Furthermore, the (initial) lack of dynamic binding of data will typically result in larger modules, require more careful consideration of the packaging of modules for run-time binding, and require explicit design attention to the role that dynamic binding will play in each specific application.

3.7 Tailored Command Environments

The new standard command environment, with EPF's and VMFA, is but one of many possible environments. It is oriented towards running programs developed under today's conventions for the treatment of static storage, with some inexpensive but powerful extensions. It is implemented in such a way that command environments tailored to different orientations can be implemented using components of the standard environment as building blocks and can coexist with processes

using the standard environment. The procedures to support a new level environment are accessible, as are the routines for mapping an EPF into address space, for allocating and initializing storage for an EPF's linkage, for re-initializing and EPF's linkage, and for reclaiming an EPF's static storage and the address space the EPF itself resides in. One could use these routines in different arrangements to build special purpose environments. For example, a transaction processing monitor might limit the environment to a single level and represent each transaction as a separate EPF and retain or discard EPF's and static storage in memory according to anticipated or observed usage patterns. A debugging environment might keep more information about routines dynamically bound so that a routine could be individually re-initialized or replaced. One might also use the environment and EPF routines exactly as the standard environment does, but change the command syntax.

4 Virtual Memory File Access

Virtual Memory File Access removes the layer of protocol and data copying that lies between a user process and data on the disk. The fundamental operation is to associate a segment address in virtual memory with a segment object in the file system. Once the association has been made, references to addresses with that segment number are references to the file contents. The only complications to this simple picture come from the fact that our file system is not just directory structures and data but is also a set of concurrency protocols, and from the mismatch between the file system's treatment of file size and the P400 architecture's treatment of segment size.

In the following treatment I use the term "file system" to refer to the disk structures and the associated lockout, access control, and quota protocols. I use the term "unit-based file system" (UBFS) for the current collection of procedures used to access the file system. "Virtual memory file access" refers to the new method of accessing the file system by mapping files and directories into a process's virtual address space.

4.1 Files as Virtual Memory Objects

In our file system a file (or sub-file of a segdir) is simply a linear array of words. VMFA views such a file as a linear array of segments all but the last in the file 64k words in length; the last segment of the file may be shorter than a full segment in length. (In other words, "no holes".) Note the distinction between a segment - a piece of data - and an address - a location data may be assigned in virtual memory. Any contiguous block of segments from a file may be assigned to some contiguous block of segment addresses in a process's address space, limited only by the availability of address space. Because different programs within a single process may access a file "concurrently" (more precisely, one program may map and reference the segment and then be suspended while another program maps and references the segment, and then the original program might be reactivated and make further references to the segment), and because a segment cannot be moved if a suspended program has knowledge of its assignment in memory, it is not good programming practice for one program to treat an object as a disconnected collection of segments where another program maps the same object into a block of addresses. Should one program map a segment from a file into an address and be suspended before completion, and another program goes to map that segment as part of a contiguous multi-segment object, and the adjacent addresses are not available, the previous mapping cannot be changed without disrupting the suspended program. This restriction should not be a significant constraint on programming style.

Once a segment has been assigned an address, that address can be used as an identifier of the segment and of the file it is part of in further transactions with the operating system. Thus, in processing a file, the name of the file need be supplied to VMFA once: as long as some segment of the file has an assigned address that address can be used to identify the file in further mapping operations.

4.2 Directories as Virtual Memory Objects

The use of an assigned address as the identifier of a segment is especially useful in the treatment of directories, which may be assigned addresses just like any other file. Once a directory has been assigned an address, that address, not the pathname, is used to identify the directory. This greatly simplifies calling sequences to VMFA - no pathnames - and provides a large number of “attach points” - as many as you care to allocate address space for - since a virtual address may serve as a shorthand identifier for a point in the file system’s directory tree.

For compatibility with the current user interface to the file system we retain the concepts of “home” and “current” directory, which may be specified in calls to VMFA by dummy virtual addresses. Otherwise, with VMFA there is no need for software to change home or current directory unless that is part of the software’s prescribed effects. Thus, software that operates on different points in the file system (like a file copy utility) need never leave the user attached away from the directory he was attached to when he invoked that software, even if the program terminates abnormally. Well-designed software will, however, incorporate a “cleanup” condition handler to remove from address space directories it has mapped into virtual memory. (More on cleanup later.)

Since directories may contain security information - passwords - and since the contents of a directory may be changed by one process while another is reading it, no ring three access rights are given to directories. Information in a directory may be read by one of two methods: the unit-based file system’s entry-at-a-time protocol and a new method introduced with VMFA which gives the entire directory contents at the time of the call. The new method allocates a segment to hold the directory snapshot and copies the contents verbatim leaving out only security information. While this may seem to violate principles of “information hiding”, any format we might choose to translate the directory contents into would only have to be translated again or otherwise processed in its ultimate use, rendering VMFA’s translation redundant. Furthermore, hitherto changes to directory format have consisted of introducing additional information into existing entry types and adding new entry types, a kind of change that the next layer of software can be made insensitive to. Format-independence can still be achieved by using the unit-based file system’s protocol or by establishing a new level of information-hiding between the VMFA snapshot and the ultimate user. This layer could incorporate other features like wild-card filters, sorting, and so on.

4.3 Opening and Closing Files

In the unit-based file system the “open” operation serves two purposes: giving the user an identifier with which to specify the file in read and write operations, and registering the user as a reader or writer for the purpose of locking out conflicting users. We have transferred both of these purposes to the “make known” operation (that is, assign an address to a segment) by using the address of a segment as its (and, in some contexts, the file’s) identifier and by defining a user to have a file open as long as any segment of the file is in his address space.

As a consequence, the user need not perform any extra operation to register himself as a user of the file for lockout purposes. On the other hand, any software which needs to take concurrent usage of files seriously must take care to keep at least one segment of the file in its address space whenever it does not want to allow another user to open it for a conflicting use. Thus when traversing a file such software will typically make known the next segment it will process before making unknown the segment it was previously using. Another strategy is to simply retain one segment as both the identifier of the file and the toehold preventing conflicting use of the file.

The possibility of use of a file in different levels of the same process dictates that VMFA keep track of redundant mapping of a segment, possibly with different modes of access. In the simplest case this requires that VMFA keep count of the number of make knowns of a segment that have not been cancelled by make unknowns. More difficult is the job of keeping track of concurrent read and write usage of a segment within a process. For example, suppose a level makes known a segment of a file for reading, that level is suspended, and another makes known the segment for writing (assuming that concurrent read and write is not forbidden). Now the process is registered as both a reader and a writer, and the user has both read and write access to the segment. Now VMFA is called to make unknown the segment. Which registration should it cancel? Since levels are treated in a nested manner, VMFA should cancel the write registration and change the access rights back to read-only. For VMFA to do this requires that it know the history of make knowns, not just the count of the different categories. We have chosen a strategy which simplifies the record-keeping of VMFA and provides an important additional benefit. A request to make known with a <u>different</u> mode of access an already known segment yields a different address assignment, (that is, the same segment will now be in two locations, one with read access, one with write access), and thus VMFA knows by the address it is given to make unknown which category of use to decrement and eventually cancel. One consequence, considered a benefit, is that a program which is interested only in reading a segment will not get write access to the address it is given for the segment even if a suspended level of its process has write access to that segment at a different address. Also a benefit is that VMFA need only keep a usage count of segments, not a history of different categories of make knowns. The one drawback is that changing access rights means changing the address of the object, or rather, that one has two addresses of the same object, one address for each access category; at worst, this might be a nuisance, at best,

it will contribute to robustness in the face of some kinds of bugs.

Closing a file is implicit in the removal from address space of all segments of the file. What should “close all” mean? Of course this is a trick question; “close all” will mean just what it means today: close all (more or less) of the unit-based file system’s file units. The question really means “what kinds of general process cleanup commands should we add with VMFA?” While we haven’t determined the final form it will take, it is clear that we wish to offer, as a minimum, the ability to totally re-initialize a process - equivalent to logging out and back in - and the ability to clear out suspended level environments without re-initializing the process environment. No doubt we will discover other useful variations of the cleanup function.

4.4 Atomic Operations

Another significant difference between VMFA and the unit-based file system is the atomicity of operations with respect to concurrent users. The indivisble operation in VMFA is the memory reference; the indivisible operation of the unit-based file system is the PRWF$$ read or write. Thus, a VMFA user of a shared segment can see a PRWF$$ operation by another process in progress. The policy we have adopted regarding this situation is to treat the two modes of usage as indepedent: unit-based file system operations act as they do today, that is, one PRWF$$ read or write is atomic with respect to another; and VMFA operations, if they are to make atomic some operation with larger scope than the memory reference, must be governed by some mechanism outside the sphere of VMFA, such as a semaphore. Like many of the design decisions of VMFA, this is a trade-off among complexity, performance, and utility. In this case we felt that most applications that require regulation of concurrent access to files rather than the simpler prevention of <u>any</u> conflicting use of files should use an appropriate regulatory mechanisms outside of the file system, since the nature and efficiency of such mechanisms are particularly sensitive to the actual concurrency regime the application must follow. For this class of applications the requirement to incorporate other mechanisms to regulate concurrent use is no imposition. The basic file-level reader/writer lock is retained with VMFA as this is necessary to support more complex mechanisms and because it suffices for the typical casual requirement for concurrency control - prevention of writing while someone is reading a file.

4.5 File Growth and File Size

There is a fundamental difference in the way the P400 virtual memory architecture treats the size of segments and the way the file system treats the size of files. The former supports segment size in units of 1024-word pages; the latter supports file size in units of 16 bit words.

Another way of characterizing the difference is that using the virtual memory architecture puts on the application the task of somehow knowing (by storing in, or associated with the objects) the size and number of objects in a segment of memory. The unit-based file system takes upon

itself some of that task in telling the application when it has reached the end of a file when reading and by setting size (on writing) with a granularity of a word.

We have reconciled these notions in the VMFA implementation by providing the minimum functions to enable the UBFS (or VMFA-based applications that interface with UBFS-based applications) to implement treatment of size to the word in a concise and efficient manner. These additional functions will not burden with additional overhead an application using files solely in a way consistent with the virtual memory architecture.

There are several differences bwteeen the two modes of access which must be reconciled in this design: the firmware does not distinguish between read and write references that cause a page fault. (It does, however, prevent writing to a write-protected segment.) Consequently, there is no easy way to tell whether a reference beyond the end of a writable segment is a write-reference and presumably intended to cause extension of the file, or is a read reference indicating some kind of error. Now is there any easy way to tell precisely how much to extend the file, that is, how many words of the page just referenced should be added to the file - once the page has been added to the file, any part of it could be referenced without another fault.

You can't even use the firmware to discover when you hit the end-of-file reading a read-only segment; it can only tell you if you have caused a page fault on the last record. This means that you can refer to a word beyond the end-of-file but within the last physical record without the firmware detecting it. It doesn't even mean that we can use the "page fault on last record" as part of a read protocol designed to warn a program when it is close to the end-of-file since the program might not cause a page fault if the file is shared with another process that happens to cause the page to be read in just before the program refers to it itself.

Another problem is that you don't know how long a file is until you look at its last record. For a DAM file this is only a nuisance: two or three disk reads will get you the answer; for a SAM file this is much worse than a nuisance: the whole file must be read to get the answer. To avoid undue overhead there must be some reasonable way to find out the required information as the file is processed to avoid a redundant traversal of the file.

The principal features of the design:

- Reference beyond the end of a segment with write access will extend it. The page fault machinery will give the new last page a size of 1024 words for file system purposes.

- Nothing reasonable can be done, so nothing will be done about detecting memory references beyond the end-of-file but still within the last record of a file. Thus, such references will not change the size recorded in the last record.

- There is a call to VMFA to set segment size within the last record of a file. VMFA-style users (and the unit-based file system implementation itself) will use it to trim a file to size for a subsequent UBFS-style user.

- There is a call to VMFA to read the state of size knowledge about a segment: the known extent of the file and a flag indicating whether reference to the next page would require (or cause, if the process has write access) extension of the file. If the latter is false then the known extent is the true size of the file. This function is used during serial access to a file to discover the end-of-file as the file is traversed, thus avoiding extra disk operations to determine in advance the file size.

- There are also an assortment of calls that yield full size knowledge of a file. These are made available to avoid the nuisance and slightly greater overhead of positioning to the end of a file to find its size.

It is important to note that there us a real distinction in treatment of size between VMFA-style use of files and unit-based file system style use of files. The size protocol is meant to allow the VMFA user and the UBFS user to use the same files, but it doesn't insure that all mixes of the two methods will always work. Errors might range from a VMFA-style user not setting the size to the word when a UBFS user expects it, rather than knowing where the end is by other means, to subtle problems with simultaneous use: the UBFS uses its N reader, 1 writer lock to prevent other UBFS users from seeing a file in transitional states between writing and setting the size. A VMFA user of a file being written can see these transitional states. The best strategy is not to mix methods of access to files that may be opened for concurrent reading and writing unless conflict is avoided by means outside the file system.

4.6 Temporary Storage and the Process Directory

Under VMFA there is no longer a distinct region of the disk reserved for segments: all paging is to segment portions of a file. Furthermore, we will encourage explicit creation of temporary segments, rather than extending the current method of allocating temporary segments by simply referring to them. Explicit creation and deletion of segments is a necessary part of the nested nature of levels: the operating system must perform the assignment of an address at run-time so that software may coexist within and between levels without interfering in static assignments of addresses for temporary storage.

The routine to create temporary segments creates the segment requested in the process directory, making up a name for the file it constitutes. A multi-segment object (i.e. a temporary object requiring multiple contiguous addresses) may be created in one call and all the segments will be in a single file. Naturally, the address assignment will be performed by VMFA, not the user. The corresponding operation to remove a temporary segment from address space deletes the segment once there are no other users of the segment. Ordinarily, a temporary object will

only be used by the process creating it, and there is no need for the process to know anything about the whereabouts of the process directory or the name of the file containing a temporary segment. Should a process need to create a temporary object that it is going to share with another process it can determine the location in the file system of the temporary file by calling the operating system and pass that to another process (as well as changing the access control on the file to enable another process to access the file). We didn't make this especially convenient to do because we felt that multiple users of a file are more easily coordinated and controlled by agreeing in advance on the location of a shared object. Better inter-process communication may change this assumption, and the design certainly doesn't preclude our making the exporting of access to a temporary object more convenient.

The location of a process's process directory is specified in a user's profile, allowing the system administrator to place these directories to distribute disk traffic, to charge the user for his process directories' use of disk space, and to govern size of the process directory with the quota system.

4.7 Networks

For the present we are proceeding on the principle that virtual memory access to files is fundamentally a local operation. Transparency with respect to networks belongs at a higher level, such as data bases or transaction processing. Network transparency will be retained in the unit-based file system. The decision to treat VMFA as a local operation is based on the difficulty of supporting remote concurrent writing of files in any sensible way at the level of VMFA. We could relax this constraint somewhat to allow transparency of remote access by VMFA for single writer versus multiple reader files, but, in general, it is more efficient to make the location of services transparent with respect to networks, rathet than the location of objects, as our experience with FAM indicates. Modification of our policy in this area should await a more completely thought-out design of a network architecture.

4.8 Miscellaneous Benefits of the Implementation

The primary benefit of the way we have implemented VMFA is that the page map of a segment is no longer tied to memory for the life of the segment. In other words, the implementation can "page out page maps." It is essential to make efficient use of wired memory allocated for page maps because VMFA will use page maps to access all file system objects. By allocating real memory for a page map only when the object is in active use we hope to increase the utilization of real memory more than enough to compensate for the larger number of segments that will be in the typical process's address space. Note that it is not only VMFA that is putting pressure on the size of a process's virtual address space: the nesting of levels and use of EPF's requires larger address spaces, and there is a general tendency simply to run bigger programs. We hope that by making the allocation of page maps a function of use, not existence, of segments that working set will be

primarily a function of the nature of reference patterns, as it should be, rather than increase as an artifact of our present fixed allocation of page maps. Observations of utilization of page maps on current systems is encouraging.

5 Miscellaneous Topics

5.1 An Alternative Way to Detect the Boundary of a Program Invocation

The method of detecting the boundary of a program invocation hinges on the discovery of the need to re-initialize a library by arranging that there be a fault when a program attempts to use it. This requires that re-initializing an EPF includes unsnapping its outward links so that every time an EPF is used within a level its outward references will be detected and bound, and the routine called tested to see if initialization is required. This also has the property that the change in the search list will be reflected in all future invocations of a program, even if it has been invoked before at the level, since all previously snapped links are undone evert time the program is invoked. While this is a desirable property, it is not the most efficient way to obtain that property if you assume that the search list rarely changes, that is, if you assume that if a particular procedure satisfied the reference the last time, it probably will next time, too. It would be more efficient to put the overhead of supporting changing a search list on the change operation, the infrequent one, rather than the link operation, the frequent one. Another property of this boundary detection method is that if there are multiple links to the same program, each will cause a fault and in turn determine whether the library needs to be initialized for the current program invocation. This puts the burden of asking whether the library needs re-initialization on the many - the callers - rather than on the few - the objects of the calls. Furthermore, the object of the call knows (in some sense) who he is, whereas the caller has to look it up somehow.

A scheme that would reverse both these choices works as follows. The first initialization of a library at a level would be detected and performed as in the current scheme: when a program attempted to link to a library its static storage would be allocated and initialized. Subsequent uses of the once-initialized library would be detected differently, however. Every entry to a library would begin with a reference to a single indirect pointer shared by all the entries in the library. The fault bit in the indirect pointer for each library that had been allocated storage at a command level would be set before every program invocation. Encountering the fault would thus indicate use of the library by a new program invocation, and the command environment would then re-initialize the static storage and turn off the fault bit. Thereafter in that invocation, any reference that had been snapped in a previous invocation to <u>any entry</u> to the library would not cause a fault on entry to the library. New links to a library require only a check as to whether the library has <u>ever</u> been initialized at the current level (which would have to be performed anyway to find the entry address): if so, just link and return - the internal reference will cause re-initialization if necessary. If there is as yet no static storage for the library at this level, allocate and initialize the library. (The first initialization could leave the fault bit off to avoid an unnecessary fault.) This method trades setting a bit in each initialized library's static storage before every command for the

alternative method's repeated searching of the library list on every new reference to a library from a library or otherwise repeatedly-used EPF retained in address space. Furthermore, it trades a memory reference (to potentially cause the fault), which can also contain the identification of the library being entered, for a table lookup. In this alternative method a change in a search list would necessitate re-initializing (or simply discarding) static storage for EPF's in completed programs, that is, EPF's not part of a current (suspended or executing) program invocation. In both schemes a change in a search order would present the possibility of inconsistent behavior of a suspended program invocation; the safest thing to do in either scheme is to recommend process re-initialization when changing the library search list except in a carefully controlled debugging environment.

The reasons we did not take this approach at this time are that it involves changes to the translators (to include the reference in every entry), which would have widened the impact and dependencies of the project; the benefits would be significant only for repeated uses of a small roster of programs and libraries; and the alternative scheme can coexist with the chosen scheme and thus be incorporated later in a compatible manner (the "PRIME Way").

5.2 Relation to Static-Mode

Static-mode programs - those programs written with static address assignments - will not require any changes to continue working as they do today. There is a slight improvement in the isolation of the effects of running a static-mode program in that, with recursive-mode libraries (which can be used by either static- ir recursive-mode programs), running a static-mode program will not interfere with a suspended recursive-mode program by re-initializing the static storage of libraries used by the suspended program. Of course, static-mode programs will continue to conflict with each other in their use of statically-assigned memory.

Static-mode programs which allocate temporary segments by simply referring to them are still a potential hazard to recursive-mode programs, but we will attempt to minimize the hazard by biasing VMFA's allocation of addresses away from the region typically used by static-mode programs today.