UIUCDCS-R-80-1005

A VIRTUAL ADDRESS KERNEL MODIFICATION FOR UNIX

by

Alfred D. Whaley

February 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Supported in part by the U.S. Department of Energy, Contract DE-AS02-76ER02383.

A VIRTUAL ADDRESS KERNEL MODIFICATION FOR UNIX

by Alfred D. Whaley
University of Illinois
Urbana, Illinois

ABSTRACT

UNIX is a modern popular high-level language operating system running principally (but not exclusively) on PDP11 computers. One of the big problems with this system is a tight restriction on kernel memory due to the 32k word virtual address space of the PDP11. The work described here divides the kernel into modules which run in independent address spaces to overcome this problem. The main software effort was to develop software tools to make it easy to work with the separate modules, and, where possible, to make the conversion without modifying the kernel source code.

Keywords: kernel, address space, operating system, UNIX, virtual memory, compatibility, minicomputer, memory limitations, software tools.

INTRODUCTION

Use of the popular Bell Labs UNIX system on a PDP11/40 can present difficulties due to the limited address space of the machine. Lowering hardware costs have caused all the major minicomputer manufacturers to enter the midsize computer market, but escalating software costs have prevented them from abandoning old instruction sets. This means the 16-bit minicomputer manufacturers (for example, Prime, Interdata, Data General, and Digital Equipment) have all been faced with the problems of using instruction sets with 16-bit address fields on machines with more memory. The PDP11 series has "solved" the problem by a memory management system which maps a virtual address space of 32K 16-bit words into a physical address space of 128K words. It is interesting to note that the early use of the virtual address concept was to provide a large virtual space with the confines of a small physical space. Little attempt has been made to solve the small virtual address problem in the PDP11 series (prior to the VAX, a substantially different computer) so, although several programs may be resident in physical memory, no single program may be over 32k bytes. Due to the dedicated use of two of eight segment registers, the UNIX kernel is restricted to 24K on a PDP11/40, 34, 23, 35, or 60, which is often inadequate. On the model 45 or 70, separate address spaces are provided for program and data, greatly alleviating the problem, although the 70 is so powerful that one tends to allocate so much buffer space in the kernel that the virtual limit again becomes a problem.

One widely distributed (though undocumented) solution to this hardware limit on the model 40 was a version of UNIX by Robert Sidebotham, faculty of Environmental Design, University of Calgary. His solution was to move the I/O buffers out of kernel address space. Unfortunately this kind of approach tends to be slow and more awkward due to the increased difficulty of accessing buffer data for filing system and other purposes. Modifications to the source code are numerous. The other difficulty with this approach is that it does not help enough. The system we were using had sixteen buffers (fairly typical) occupying 4k words, which is therefore the maximum amount of address space that could have been regained with the Calgary method.

Typical UNIX additions of many I/O drivers or code to handle interprocess communications can easily take up more than 4K-words, and cannot be accommodated by this technique.

Another system which solves the problem of address space is MERT[1]. MERT is a full fledged multiprocessing environment which runs UNIX as one possible subsystem. It is reputed to be quite slow compared to standard UNIX, not surprising considering its elaborate nature, but it appears to have the desired features. MERT is inappropriate for a PDP-11/40 because it requires the split instruction and data space of the PDP-11/45 or PDP-11/70.

The final motivating force for the virtual kernel development described here was a multiplexor [2] which allows a large number of input and output lines (up to 256) to be connected, each able to run at moderately high speeds. The amount of tabular information for software control of a UNIX terminal or other I/O line is fairly large; with so many lines we quickly ran out of kernel space.

The goals of this project were to provide an extremely general facility which would make it easy to break the kernel into resident overlays so that not all of the executable code is addressable at any one time. The techniques developed resulted in a set of software tools rather than being tied to some peculiarity of the UNIX kernel. As it turned out, with the correct tools developed, there was a particular division of the kernel into virtual modules which involved an extremely low amount of work.

The next section treats the problems inherent in and characteristics of the UNIX kernel that affect any attempt to split it up into multiple virtual segments. Following is a section which describes the software tools developed for the project.

ISSUES OF SEPARATION

Division of a program into overlays creates one primary problem, that of reference to addresses that are not currently valid. It is useful however to treat text (executable code) and data addresses separately. Data addresses are only a problem if they are referenced by text which is not in their overlay, so all data referenced by more than one segment has been moved into a low core common non-overlay segment that is always addressable. This solution is particularly suitable for UNIX since the data which is addressed in many different routines is almost entirely in a small number of large arrays. These arrays are the central storehouse of information on processes, files, and the like, and their use is so widespread that easy access by all overlays is essential.

If shared data could not all be in the common area, it could, in isolated instances, be accessed not directly but by invoking a subroutine in a foreign overlay segment which could address the data desired. This would involve modifying the source code slightly to change from a data reference to a procedure call. Another way to handle data is to overlay it directly, which requires that each reference to the data involved be preceded by a call to the overlay manager to make sure that the segment is addressable. This approach would be useful for UNIX on a PDP11/70 where the amount of buffer space alone could be over 32K. Fortunately, on the PDP11/40, both these methods may be avoided as all the arrays involved fit easily into the common segment. The problem of accessing data may be therefore conveniently bypassed without overlays, allowing us to focus our attention in the remainder of this section on the problem of text.

The fact that text is broken only at procedure boundaries when creating overlay segments, reduces the text problem to one of procedure linkage across overlay segment boundaries. References in the source code to foreign procedures are converted to calls to an overlay manager with the procedure name and original calling arguments and the numeric designation of the new overlay as arguments. Since our policy is to avoid modifying the source code, this modification is done through the use of macros. The overlay manager switches the hardware segment registers to make the new overlay addressable, and calls the procedure. On return, the manager restores the addressability of the original caller and returns. As an example the subroutine subr is not in the current overlay so the call

```
        subr (a, b, c);
```

is converted by a macro to

```
        vswtch (3, a, b, c, subr, 1);
```

where 3 is the argument count and 1 is the overlay containing subr.

Since the old kernel did not need to be too much larger than the size of the address space, any split should produce an acceptable arrangement. For this reason only one additional segment was originally created; it contains the non-disk drivers--see Figure 1. The two segments are referred to as the main and device segments.

In order not to slow down any of the usual functions, routines implementing system calls and multiprocessing functions are in one segment, so that nearly all normal functions on a system call may be performed without overlay mapping changes. Drivers for non-disk I/O are generally involved in activities consisting of large real-time delays, so the division seems fairly natural. This means that most normal timesharing activities may be carried out as in the original UNIX kernel without reloading the segment registers. The typical user does not create any additional delay except during I/O to a terminal.

A third overlay segment in Figure 1 is the "begin" segment. Initialization code is placed here and then freed just before exit so that the storage segment occupied is available for user programs.

Although the principle problem with overlays was presented at the start of this section as being the validity of addresses, a few peculiar difficulties exist which may not be placed in this category. They are briefly summarized here. A principle consideration with virtual addresses is that I/O equipment does not (on this machine and most others) go through the segmentation hardware. The physical rather than virtual address must be calculated for use in I/O. The large I/O buffers have already been described as being in the common section, where physical and virtual addresses are the same and for the same reason the "clists", (16-byte terminal I/O buffers) have also been left in common. These economies mean less changes (mostly none in fact) in the I/O drivers. Only one driver with some canned messages in its own data area needed modification. A routine in the common area called physad returns the physical address of a virtual address for this purpose.

A problem that we have been able to ignore is that of sleep addresses. Sleep and wakeup are a pair of synchronizing primitives analogous to P and V, that use an address as a unique event identifier. The address is never actually referenced but must be unique. Since the segment identifier is not included (it could be), the virtual addresses of different sleep/wakeup pairs in different segments could be duplicates when not intended. Such a coincidence is extremely unlikely and due to UNIX structure such a coincidence is not fatal, merely inefficient (since "false" wakeups are already possible, and all routines are required to check validity and re-sleep if necessary), so the problem has been ignored.

SUPPORT TOOLS

loader

The principle ingredient in this system is a modified relocating loader which generates virtual segments after resolving external references. A fair amount of thought was necessary here, because it was not desired to make extensive modifications to the c compiler (and language) and object file format for the sake of this one project. In conflict with this was the imperative need for certain kinds of communication of a static predefined nature between the various virtual segments. The major approach taken here was to define a set of new reserved external variable names that act as keywords. Some of these are just special symbols such as the symbol PHEND which is set to the value of the physical ending address of used memory. Other symbols are operators. By way of illustration, in the statements

```
        extern sqrt, PHEND;
        int var sqrt;
        int xyz PHEND;
```

the variable var is initialized to the virtual address of the sqrt routine. The value is filled in by the loader. The variable xyz is initialized not to the address of the external item PHEND, but the physical address of the first free memory after all overlay segments.

Reserved names used as operators may be regarded as unary operators applied to the external reference seen most recently by the loader. In the above example there is no guarantee that the loader will see variables var and xyz in the precise order coded, so for operators, use of an array is prudent. For example in the array:

```
int array(2) {
        sqrt,
        PHYSADR
};
```

there are two words defined, the first having the address of the square root routine, and the second having the physical address of the start of not the square root routine, but the virtual segment that it is contained in. Such information is valuable in a mechanism that switches the values in the segmentation registers to change which segment is current, but is of little use elsewhere. Clearly external names have scope over all segments.

Referring again to Figure 1 we can see how a simplified UNIX kernel may be assembled by the following single loader command:

```
ld -o modcom vswtch.o -v 0
   -o modmain pe.o pa.o -v -1
   -o moddev pd.o pb.o -v 60000
   -o modbegin pc.o -v 60000
```

This command causes the loader to output to file modcom all object files before the first -v flag, i.e. object file vswtch.o containing procedure vswtch. The parameters "-v 0" specify the starting virtual address of the overlay module; in this case modcom will start at virtual address zero. As the first module generated, it also starts at physical address zero. Next the output name modmain is assigned with the next -o parameter. Object files pe.o and pa.o will be placed in this module. The "-v -1" parameters assign the starting virtual address. In this case, since the numeric field is preceded by a minus sign, the number 1 is taken as being another module (in this case modcom) after which this module (modmain) should follow compactly in the assignment of virtual addresses.

During execution, exactly one of the three modules moddev, modbegin, and the portion of modmain above virtual address 60000, will be addressable at any one time. They cannot all be addressable since they all occupy the same part of the address space. Subroutine vswtch in modcom controls the contents of the hardware virtual address registers, and thus which module is addressable. These three modules are therefore overlay segments, while modcom is "resident." When procedure pa calls procedure pb (the call is illustrated in Figure 1 after macro expansion), control is actually passed by procedure call to vswtch. The last argument in the call is a 3 which vswtch uses to index into array segadr. Array element 3 is the value of PHYSADR (in the language C, indexing starts at zero) which immediately follows the address of pb (in segadr(2)). As explained earlier, the loader will have placed in segadr(3) the physical address of moddev. Vswtch then uses this value to effect the change in virtual address mapping.

For convenience, each module (modcom, moddev, etc.) has the identical format of former object files except that an unused word in the header tells the physical address at which the segment is to be loaded. This address is assigned by the loader. This compatibility means that the utility programs for dealing with object files still have some restricted use. The size and name list programs still work, and the debugger could be made to work easily by subtracting the virtual offset when accessing the object file.

In order to have the convenience of a single composite file, the standard UNIX ar command is used to build an archive file with the four segments. An archive file is in essence a subroutine library in a single file. An archive command suitable for the above example would be:

```
ar r /unix moddev modcom modmain modbegin
```

which would create an archive file of name "/unix" with four members in it, moddev, modcom, modbegin, and modmain. The bootstrap program is capable of recognizing an archive file, and of loading each member separately.

The last consideration has to do with the text and data areas.

Traditionally, UNIX programs are split up into text and data areas with an additional bss area that is uninitialized data (and only its size is carried in the object file). In the new scheme, each segment must have its own text, data and bss areas, since it is impossible to execute a segment that cannot access its data. This is the reason in the above example that modcom and modmain could not be put in one segment by the loader, even though they are contiguous.

In order to verify more easily that the program being loaded makes no undesired explicit cross-segment references, an optional diagnostic printout lists such references. In addition to these modifications to the UNIX loader, it has been improved in other ways. The main change is that it now stops searching libraries when all external references are resolved. Because it has been possible to use the same loader for regular timesharing use as well as forming UNIX kernels without a lot of awkward switches, the searching efficiency has meant much to the regular user of the program.

bootstrap

Previously, UNIX was loaded by a small assembly language program that occupied the first block of some device, usually a disk pack. That program, when executed, allowed the operator to type a single name of the file containing the program to be loaded--usually the UNIX kernel. The new virtual kernel is so large that no simple program that does not utilize the virtual memory hardware can load it. The traditional boot program therefore now loads without user query another program which acts more or less the same operationally, but has the capability to load any old style program (stand alone or unix kernel), or a new large kernel. It examines the header to determine whether the file is a single object file or an archive file with several object files. Each object file found is loaded at the physical address given in the object header. The names of the archive members are ignored. This is a nice program that has all the features one could never fit in a boot program before, but is not very interesting theoretically. It is noteworthy however, that the program is not tied to any particular form of the UNIX kernel. It would be a useful tool for loading the UNIX kernel if the complete virtual structure of the kernel were altered. In an experimental research or classroom environmental this freedom would be useful.

C preprocessor

There is a preprocessor for the C compiler which handles definition of tokens for character string substitution later in the source and also provides a very simple macro facility. As one of the goals of this virtual conversion is not to change code any more than possible, calls to subroutines in foreign overlay segments are converted via the macro facility to calls to the vswtch routine (as discussed earlier), for example

```
        #define subr(x, y, z)   vswtch (3, x, y, z, %subr%, 1)
```

Since the preprocessor repeatedly rescans a line until all macros are found, an excape mechanism was introduced, where the characters %...% make the second subr invisible.

Organizationally, all subroutines in an overlay segment are placed in the same file directory, and a single file called macs with all macros needed for those routines is included. Therefore, the only change to most of the routines in the UNIX kernel was the inclusion of the single line:

```
        #include "macs"
```

(#include is a standard preprocessor feature). In order to move a program to a different overlay segment, it need merely be moved to the appropriate directory, with no source changes.

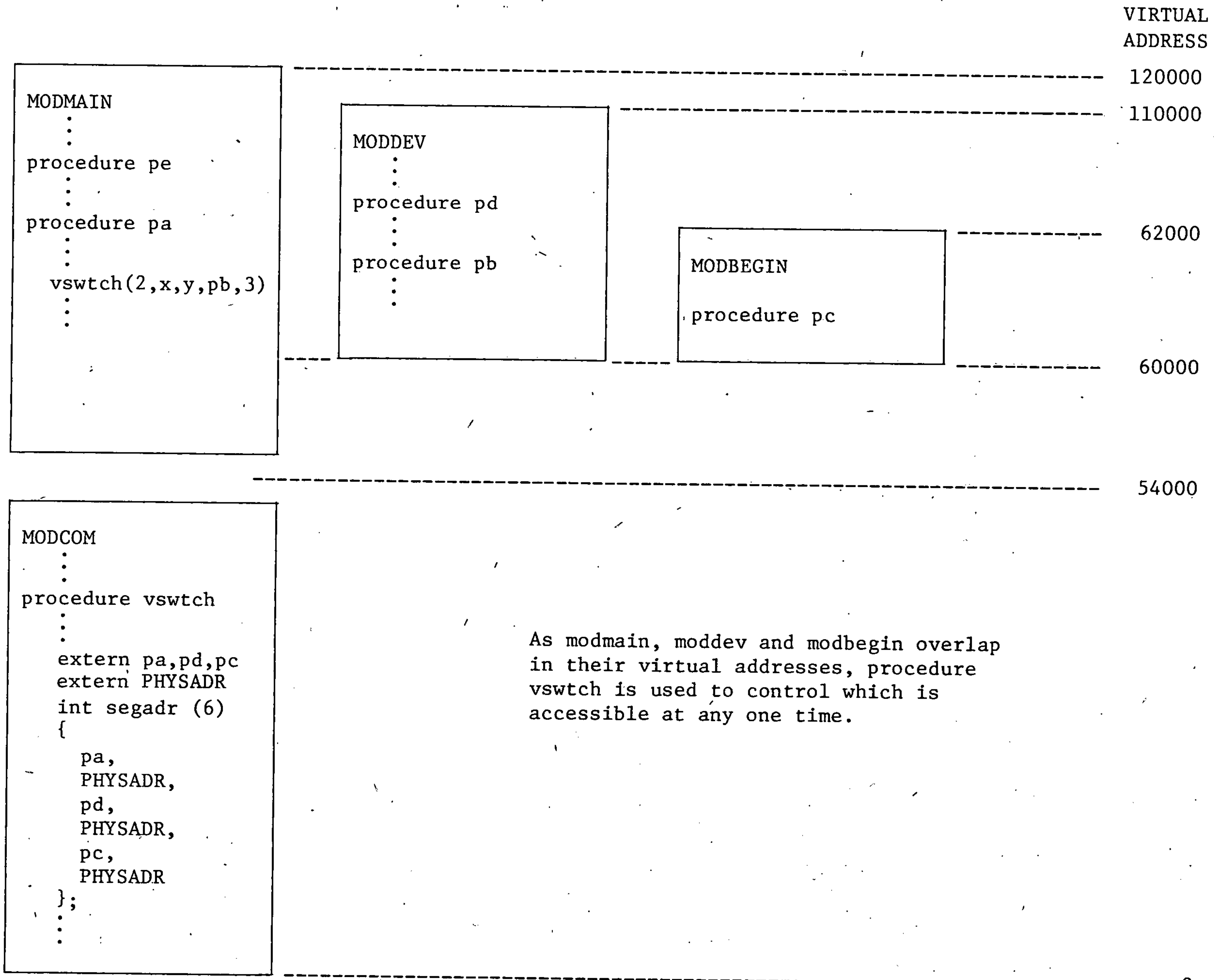
CONCLUSION

A set of techniques have been described which have an elegant simplicity and yet a flexibility that should make them valuable in a research environment. Excellent success was obtained in the low impact on the source code, and in avoiding new languages and data types to accomplish these goals. We believe the tools developed are at least as valuable as the fact of an easily expandable kernel, which itself has been an urgent need on our and other installations. Version 7 UNIX can not even be run on a PDP11/40 or PDP11/23 (except in an extremely restricted single user system) without the modifications developed here. The kernel itself works reliably and does not subjectively seem to be slower than the original.

REFERENCES

[1] Bayer, D.L. and H. Lycklama, "MERT - A Multi-Environment Real-Time Operating System," ACM Operating Systems Review 9 (5), 1975, 33-42.

[2] Whaley, A.D., "A PDP11 Multiplexor," Dept. Computer Sci. Rpt. UIUCDCS-78-942, Univ. Illinois at Urbana-Champaign, 1978.



```
MODMAIN
    :
procedure pe
    :
procedure pa
    :
  vswtch(2,x,y,pb,3)
    :
```

```
MODDEV
    :
procedure pd
    :
procedure pb
    :
```

```
MODBEGIN

procedure pc
```

```
MODCOM
    :
procedure vswtch
    :
   extern pa,pd,pc
   extern PHYSADR
   int segadr (6)
   {
     pa,
     PHYSADR,
     pd,
     PHYSADR,
     pc,
     PHYSADR
   };
    :
```

As modmain, moddev and modbegin overlap in their virtual addresses, procedure vswtch is used to control which is accessible at any one time.

Figure 1. Schematic map of UNIX kernel with four virtual segments.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1005	2.	3. Recipient's Accession No.
4. Title and Subtitle A VIRTUAL ADDRESS KERNEL MODIFICATION FOR UNIX			5. Report Date February 1980
			6.
7. Author(s) Alfred D. Whaley			8. Performing Organization Rept. No. R-80-1005
9. Performing Organization Name and Address Department of Computer Science University of Illinois U-C Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. DE-AS02-76ER02383
12. Sponsoring Organization Name and Address U.S. Department of Energy Chicago Operations Office 9800 South Cass Avenue Chicago, IL 60439			13. Type of Report & Period Covered technical
			14.
15. Supplementary Notes			

16. Abstracts

UNIX is a modern popular high-level language operating system running principally (but not exclusively) on PDP11 computers. One of the big problems with this system is a tight restriction on kernel memory due to the 32k word virtual address space of the PDP11. The work described here divides the kernel into modules which run in independent address spaces to overcome this problem. The main software effort was to develop software tools to make it easy to work with the separate modules, and where possible, to make the conversion without modifying the kernel source code.

17. Key Words and Document Analysis. 17a. Descriptors

kernel
address space
operating system
UNIX
virtual memory
compatibility
minicomputer
memory limitations
software tools

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 16
	20. Security Class (This Page UNCLASSIFIED	22. Price