Porting UNIX to the 386: *A Practical Approach*

Designing the software specification

William Frederick Jolitz and Lynne Greer Jolitz

The University of California's Berkeley Software Distribution (BSD) has been the catalyst for much of the innovative work done with the UNIX operating system in both the research and commercial sectors. Encompassing over 150 Mbytes (and growing) of cutting-edge operating systems, networking, and applications software, BSD is a fully functional and nonproprietary complete operating systems software distribution (see Figure 1). In fact, every version of UNIX available from every vendor contains at least some Berkeley UNIX code, particularly in the areas of filesystems and networking technologies. However, unless one could pay the high cost of site licenses and equipment, access to this software was simply not within the means of most individual programmers and smaller research groups.

The 386BSD project was established in the summer of 1989 for the specific purpose of porting BSD to the Intel 80386 microprocessor platform so that the tools this software offers can be made available to any programmer or research group with a 386 PC. In coordination with the Computer Systems Research Group (CSRG) at the University of California at Berkeley, we successively ported a basic research system to a common AT class machine (see Figure 2), with the result that approximately 65 percent of all 32-bit systems could immediately make use of this new definition of UNIX. We have been refining and improving this base port ever since.

By providing the base 386BSD port to CSRG, our hope is to foster new interest in Berkeley UNIX technology and to speed its acceptance and use worldwide. We hope to see those interested in this technology build on it in both commercial and noncommercial ventures.

In this and following articles, we will examine the key aspects of software, strategy, and experience that encompassed a project of this magnitude. We intend to explore the process of the 386BSD port, while learning to effectively exploit features of the 386 architecture for use with an advanced operating system. We also intend to outline some of the tradeoffs in implementation goals which must be periodically reexamined. Finally, we will highlight extensions which remain for future work, perhaps to be done by some of you reading this article today. Note that we are assuming familiarity with UNIX, its concepts and structures, and the basic functions of the 386, so we will not present exhaustive coverage of these areas.

In this installment, we discuss the beginning of our project and the initial framework that guided our efforts, in particular, the development of the 386BSD specification. Future articles will address specific topics of interest and actual nonproprietary code fragments used in 386BSD. Among the future areas to be covered are:

- MS-DOS utilities for beginning to port 386BSD
- 386BSD process context switching
- Executing the first 386BSD process on the PC

Prior to leading the 386BSD project, Bill was the founder and CEO of Symmetric Computer Systems, a BSD-based workstation and networking products manufacturer. He was the principal developer of 2.8 and 2.9 BSD and the chief architect of National Semiconductor's GENIX project, the first virtual memory microprocessor-based UNIX system. Prior to establishing TeleMuse, a market research firm, Lynne was vice president of marketing at Symmetric Computer Systems. She has produced white papers on strategic topics for the telecommunications, electronics, and power industries. Bill and Lynne conduct seminars on BSD, ISDN, and TCP/IP, and are in the process of producing a book on 386BSD and a textbook focusing on the applications layer of the Internet Protocol Suite. They can by contacted via e-mail at william@berkeley.edu or at uunet!william.

- 386BSD kernel interrupt and exception handling
- 386BSD INTERNET networking
- ISA device drivers and system support
- 386BSD bootstrap process

Getting Started: References, Equipment, and Software

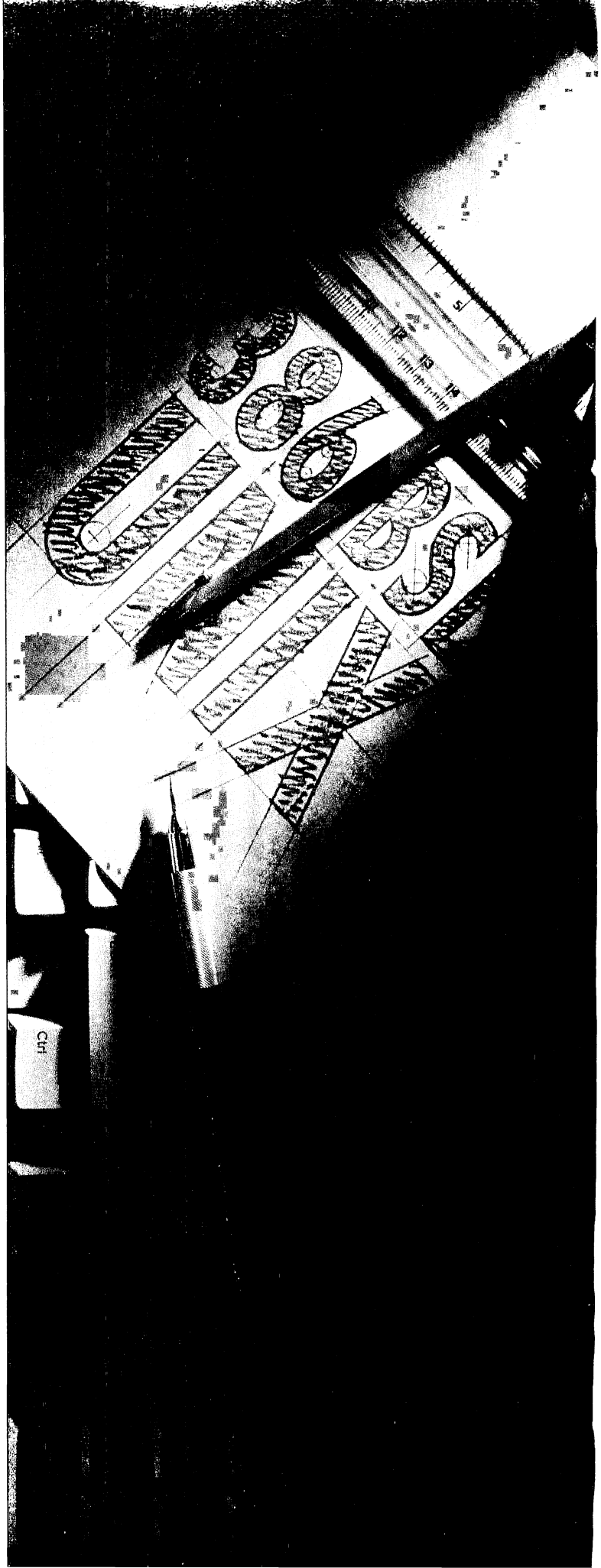
Most software ports begin with the naive assumption that the UNIX kernel is merely a C program with a handful of functions, supporting other utility C programs on demand. While in essence this is true, in practice this is a vast oversimplification. Nevertheless, in the tradition of great projects, we acquired a few tools and other items before getting down to work:

- *The Design and Implementation of the 4.3BSD UNIX Operating System* by Leffler, McKusick, Karels, and Quarterman (Addison-Wesley, 1989) and *Programming the 80386* by Crawford and Gelsinger (Sybex, 1987) were purchased from a bookstore in Berkeley. Since *no one* on our team possessed any extensive technical background on either the 386 or the IBM PC, the 80386 book was our sole resource for the microprocessor. The 4.3BSD book illuminated some of the obscure areas and requirements of the BSD UNIX operating systems kernel. We highly recommend these books. Both books have become somewhat shopworn during the process — the 80386 book has had it's covers taped twice, primarily due to being thrown repeatedly across the room in the general direction of the trash can. This book, while the best resource available on the subject, is not as complete as one might hope, primarily because the 80386 is a complex animal and is enigmatic in the correct use of its many features. Segmentation exception handling descriptions should not be taken literally, although the book was of great value when writing the first versions of exception handling code. Some portions of the software were even determined empirically. (Intel was not eager to provide any information.) The single biggest problem encountered in our project was that of inadequate 80386 documentation.
- A completely blank, inexpensive standard 386 AT clone was the selected hardware platform. To minimize expenses and to emphasize commonality, we chose to support only the basic 386 platform.
- Using exploratory programs written in Borland's Turbo C, we were able to explore the typical AT hardware. These exercises permitted us to better understand the information contained in IBM's *Technical Reference Guide Personal Computer AT*, a classic if not obscure work. We then tested the mechanisms inside the AT to make certain we knew what must be provided in order to generate the necessary software driver support for BSD UNIX.
- Our initial kernel source was the 4.3BSD Tahoe release (available for an obscure machine, the CCI Power 6/32, and as similar to the 386 as a can opener), at that time the most stable and recent release.

All of these references and the equipment were examined prior to generating even the first line of code. An understanding of the architectures of the hardware and software is critical to developing an appropriate 386BSD specification. Thus, we were able to ensure a successful port, even when unanticipated problems arose.

Development of the 386BSD Specification

Once all the materials were gathered, the temptation was to immediately sit at the PC and write code. This is a temptation that should always be vigorously avoided. One needs to sit down and carefully break down this project into smaller bites. However, because many parts of this project



are interrelated, we must insure that the internal standards are uniformly maintained by all areas of the port and during all phases. In other words, the bridge must meet in the center.

Therefore, instead of plunging directly into development, we began the most critical phase of this (or any) port — that of creating the 386BSD specification. This specification addressed the following major issues:

- Segmentation and paging
- Virtual and physical address space
- Process context description
- System call interfaces
- ISA device requirements
- Microprocessor idiosyncrasies
- Bootstrap

Unlike a commercial specification, the 386BSD specification was intended to be lightweight and flexible. We wanted to focus 386BSD without making the specification a major work in itself. We also knew that many of the finer points would change as we got closer to our goal.

The Definition of the 386BSD Specification

At first glance, the choice of the 386 microprocessor and ISA system architecture appears to define the operating system's machine-dependent requirements. For example, on the original 8088 PC to the present, MS-DOS would use the software interrupt INT $XX instruction to dispatch through the interrupt vector table entry XX, and then dispatch to the desired system call inside MS-DOS. This was the only way application programs could call the operating system.

Had this regularity been true for the UNIX operating system, all 80*x*86 UNIX systems would be alike, and the development of a specification would be a simple task. However, in exploiting the power and flexibility of UNIX, one is faced with a grander specification. The kernel architect is now faced with competing alternatives. With UNIX, the choices are no longer "cut and dried."

Adding to this dilemma, the 386 is at least two generations beyond its simple ancestor. The enhanced features the 386 now offers allow us many competing ways to satisfy a UNIX system design. Continuing our example, instead of using the INT $XX instruction, we can use the intersegment LCALL instruction to call the operating system through call gate segments. We can use some powerful features of the 386, but at the cost of a more elaborate mechanism. Is it worth it?

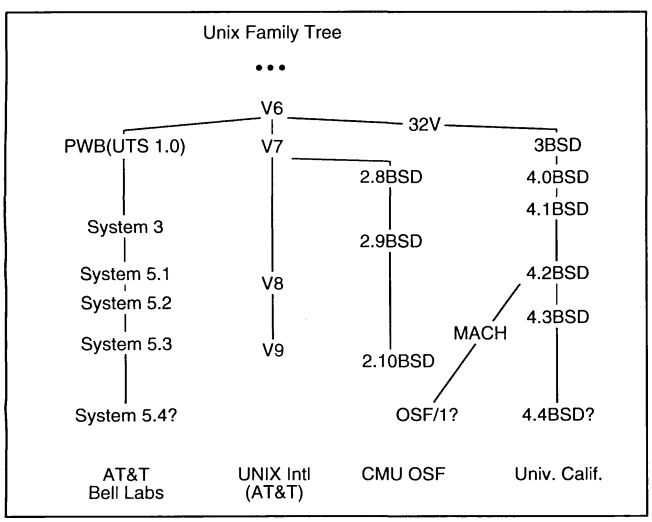


Figure 1: *The UNIX family tree*

In this case, the LCALL instruction can be used to support reverse compatibility with other versions of UNIX in the form of an applications package rather than within the operating systems kernel, and thus may be worth the effort. However, choosing among the myriad, often conflicting, alternatives is typically a task fraught with peril.

For the 386BSD project, we first determined our priorities: *100 percent BSD kernel and user functionality.* The system must contain all important underlying mechanisms of the Berkeley UNIX system. Any extensive modification pertaining to how Berkeley UNIX functions on other extant platforms can result in incompatibility. Incompatibility is like an irritating insect that bites in many places — and tends to lay hidden until after extensive distribution. As such, we did not exploit some features of the 386, such as its elaborate segmented architecture, at the expense of incompatibility.

Efficient use of the native processor architecture. We would like to use the system in ways to obtain the highest performance and greatest functionality possible.

Interoperability with existing commercial standards. We would like to use the system in ways which maintains compliance with extant commercial standards. We do not intend to unnecessarily create arbitrary new standards if current standards are acceptable.

Rapid implementation of the basic operating system. One maxim of any UNIX development effort is "the best tool to build a UNIX system IS a UNIX system." We needed to bootstrap ourselves rapidly into operation and leverage 386BSD itself to complete the project.

Conflicts in Priorities

These basic priorities inherently conflict. For example, BSD systems have basic incompatibilities with the AT&T System 5 UNIX systems, because each project has firm interests and no compelling need to cooperate. As such, perfect compatibility is impossible to achieve given our project focus. The opposite tact, no compatibility constraints, is also not completely acceptable, because we are dealing with the PC class of computers and not minicomputers or workstations. Fine grain differences also exist among the many standards currently competing for favor in the world of 386 UNIX systems.

386BSD Port Goals: A Practical Approach

Given all of these trade-offs, we decided to take what we call a "practical" approach to 386BSD. We concentrated primarily on "hard adherence" to both BSD operability and high-performance implementation, for the simple reason that 386BSD is a research project intended for use by the research community. However, because even this audience depends on commercial resources, we decided to invest some of our effort in the development of a few fundamental

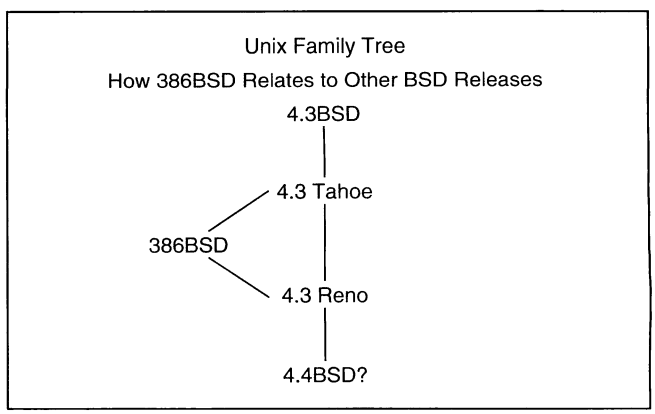
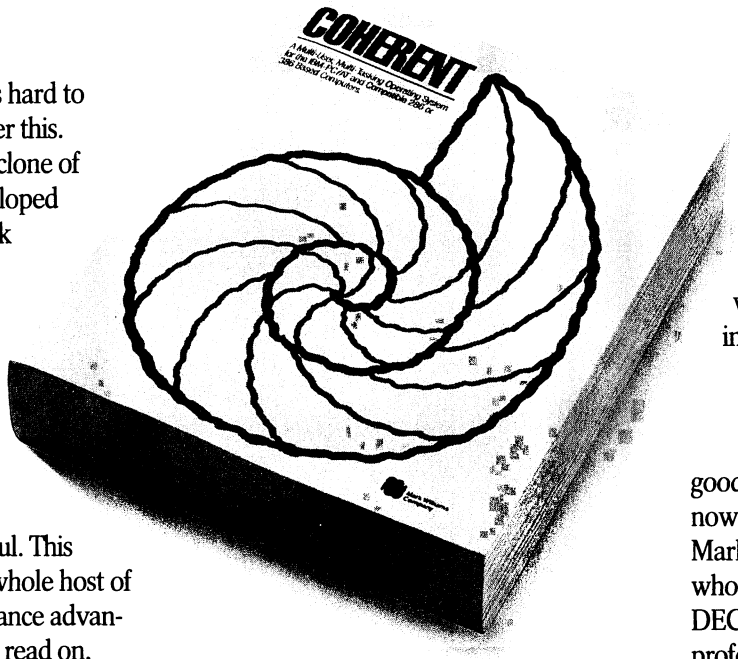


Figure 2: *386BSD and other BSD releases*

Everything You Ever Wanted In UNIX. And Less. $99.95.*



OK. We know it's hard to believe. So just consider this. Coherent™ is a virtual clone of UNIX. But it was developed independently by Mark Williams Company. Which means we don't pay hundreds of dollars per copy in licensing fees.

What's more, Coherent embodies the original tenet of UNIX: small is beautiful. This simple fact leads to a whole host of both cost and performance advantages for Coherent. So read on, because there's a lot more to Coherent than its price.

SMALLER, FASTER...BETTER.

Everybody appreciates a good deal. But what is it that makes small so great?

For one thing, Coherent gives you UNIX capabilities on a machine you can actually afford. Requiring only 10 megabytes of disk space,

LESS IS MORE!	Coherent For the IBM-PC/AT and compatible 286 or 386 based machines.	Santa Cruz Operation's XENIX 286, Version 2.3.2
No. of Manuals	1	8
No. of Disks	4	21
Kernel Size	64K	198K
Install Time	20-30 min.	3-4 hours
Suggested Disk Space	10 meg	30 meg
Min. Memory Required	640K	1-2 meg
Performance*	38.7 sec	100.3 sec
Price	$99.95	$1495.00

*Byte Exed benchmark, 1000 iterations on 20 MHZ 386.
Hardware requirements: 1.2 meg 5¼" or 1.4 meg 3½" floppy, and hard disk. SCSI device driver available soon. Does not run on Microchannel machines.

Coherent can reside with DOS. So you can keep all your DOS applications and move up to Coherent. You can also have it running faster, learn it faster and get faster overall performance. All because Coherent is small. Sounds beautiful, doesn't it?

But small wouldn't be so great if it didn't do the job it was meant to do.

EVERYTHING UNIX WAS MEANT TO DO.

Like the original UNIX, Coherent is a powerful multi-user, multi-tasking development system. With a complete UNIX-compatible kernel which makes a vast world of UNIX software available including over a gigabyte of public domain software.

Coherent also comes with Lex and Yacc, a complete C compiler and a full set of nearly 200 UNIX commands including text processing, program development, administrative and maintenance commands.

And with UUCP, the UNIX to UNIX Communication Program that connects you to a world-wide network of free software, news and millions of users. All for the cost of a phone call.

We could go on, but stop we must to get in a few more very important points.

EXPERIENCE, SUPPORT AND GUARANTEES.

Wondering how something as good as Coherent could come from nowhere? Well it didn't. It came from Mark Williams Company, people who've developed C compilers for DEC, Intel, Wang and thousands of professional programmers.

We make all this experience available to users through complete technical support via telephone. And from the original system developers, too!

Yes, we know $99.95 may still be hard to believe. But we've made it fool-proof to find out for yourself. With a 60-day money-back no-hassles guarantee.

You have to be more than just a little curious about Coherent by now. So why not just do it? Pick up that phone and order today.

You'll be on your way to having everything you ever wanted in UNIX. And for a lot less than you ever expected.

1-800-MARK WMS

(1-800-627-5967 or 1-708-291-6700)

60-DAY MONEY BACK GUARANTEE!

Mark Williams Company
60 Revere Drive
Northbrook, IL 60062

*Plus shipping and handling. Coherent is a trademark of Mark Williams Company. UNIX is a trademark of AT&T. XENIX is a trademark of Microsoft.

(continued from page 18)

areas such as System Call Interface Definition.

By dealing with these basic areas, we allowed for limited adherence to commercial standards from the start, with the ability to gradually extend 386BSD as needed. (For example, in future releases we hope to offer some degree of support for segmentation and VM8086 mode.) We have also tried, when possible, to conform to the spirit of the 386 Application Binary Interface (ABI) and its predecessor Binary Compatible Standard (BCS) when they did not conflict with our adherence to Berkeley UNIX.

The 386 paging mechanism impacts the 386BSD specification with respect to address space allocation constants: Each page is 4 Kbytes in size and must reflect the minimum granularity of address space allocation, while each page of page tables maps 4 Mbytes of address space

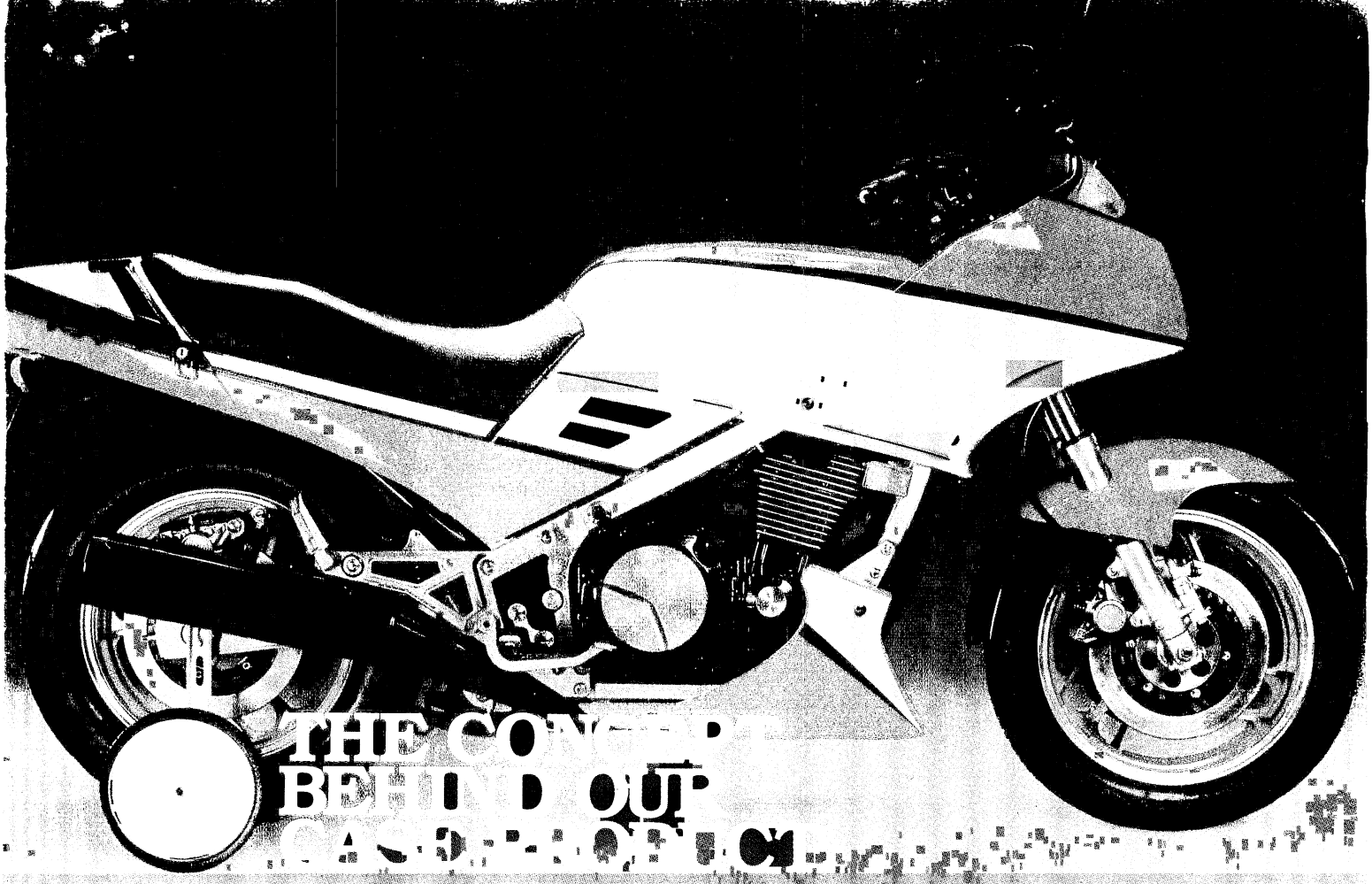
Some may take issue with this stance, seeing binary compatibility standards entirely as an "all or nothing" issue. Those who spend a great deal of time arguing over the big end and the little end of the ABI egg are usually involved in maintaining control over the shrink-wrap commercial software market. However, those who wish to ignore the ABI juggernaut are also ignoring the largest body of UNIX software outside the research community. In this case, ignorance is simply a mask for arrogance. As we stated earlier, we have tried to take a "practical" approach that builds in the flexibility without altering the scope of our project.

Many people wonder why UNIX systems are so big and complex. A look through any UNIX kernel can quickly answer this question. Many different groups prefer to further standard agenda by claiming a piece of the kernel for their own use, instead of redesigning it for common support or moving things out of it that really belong in an application process. SVR4 alone is rumored to contain 14 different filesystems which are just a variation on a theme. This "Chinese menu" approach to kernel design has resulted in a bloated kernel that is difficult to enhance or maintain. Because standards by accumulation just don't work, with 386BSD we strive to avoid such nonsense.

Another goal of our project was to insure that all code developed for the 386-specific portions of this project be unique and novel. This is to prevent any particular commercial agent from arbitrarily appropriating, monopolizing, or prohibiting discussion and distribution of this code. This is the major reason why we are able to examine some of the interesting mechanisms of 386BSD without the censorious effect of proprietary license agreements.

Microprocessor and System Specification Issues

Our specification required that we break it down into two basic technical areas: the microprocessor itself and the sur-



System Architect has the power to handle your most complex applications. And it's so easy to use, even beginners will be productive in no time.

"The software's incredible ease of use belies the power hidden within." **Computer Language**

System Architect works with such methodologies as DeMarco/Yourdon, Gane & Sarson, Ward & Mellor (real-time), entity relation diagrams, decomposition diagrams, object oriented design (optional), state transition diagrams, and flow charts.

Normal-ization

RELEASE 2.1

Super type/ Sub type

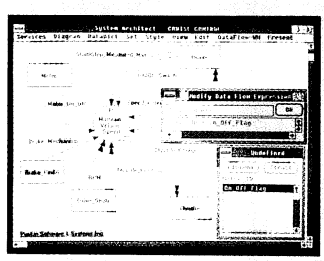
Automated documentation

Multiple methodologies

User-defined attributes

Requirements traceability

Rules & balancing



SQL-like custom Reporting

Network version available

Matrix reporting

Integrated data dictionary

Import/export capability

Auto leveling

System Architect is a pleasure to work with. It's Windows-based, has context-sensitive help, and a novice mode.

"SA is an excellent value." ***CASE Trends***

At **$1,395,** System Architect is quite affordable. And it runs on almost any PC.

"...truly a price performance leader." **System Builder**

For a powerful CASE product that's easy to use and affordable, look to System Architect. It's the right concept for CASE.

FOR MORE INFORMATION, CALL (212) 571-3434

"System Architect stood out from many other prospects because it had the best core technology." **Toshiba Corporation**

With System Architect, you get support for an integrated data dictionary/encyclopedia, and multi-user support both with and without a network. And System Architect's open architecture lets you easily import and export data to other products.

"We're surprised with its flexibility and much taken with the idea of being able to link different kinds of diagrams..." **Cutter Information's CASE Strategies**

POPKIN
Software & Systems Inc.
11 Park Place, NY, NY 10007
(212) 571-3434
Fax: (212) 571-3436

SystemArchitect™

MICROSOFT WINDOWS
Version 3.0 Compatible Product

Supporting IBM's AD/Cycle

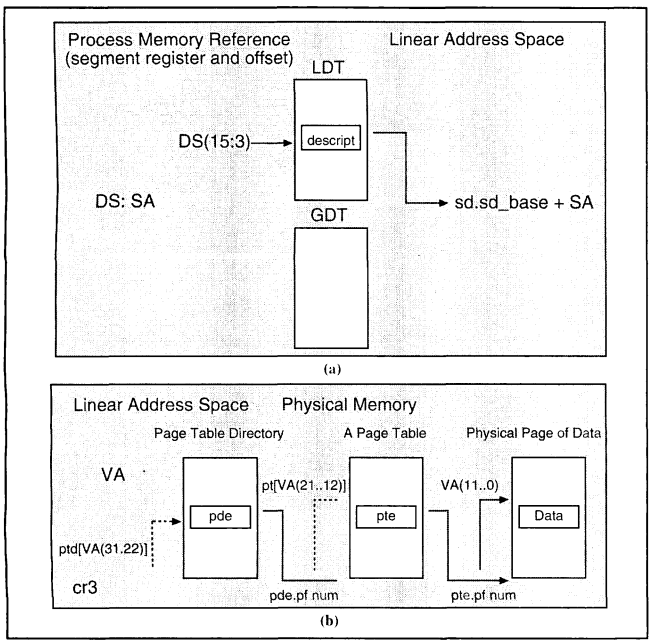


Figure 3: *386 memory management: (a) segementation (b) paging*

(continued from page 20)
rounding system hardware. In keeping with our goals, we segregated the two in order to allow future support for other buses (such as EISA and Micro Channel) and to avoid obscuring microprocessor issues.

The microprocessor required much delineation in the areas of segment and paging strategies, virtual memory allocation and other memory management issues, communications primitives, context switching, faults, and the system call interface. We also had to factor in microprocessor idiosyncrasies and bugs as we went along. On the system side, we concentrated on ISA bus considerations.

We first outline some of the major issues revolving around the 386 microprocessor itself and how they relate to a Berkeley UNIX port.

386 Memory Management Vitals

Most popular microprocessors use either segmentation or paging to manage memory address space access. The 386 is rare in that it possesses both. In fact, since segmentation, (Figure 3(a)), is placed on top of paging (Figure 3(b)), you are expected to use segmentation in some form any time memory is paged. And, most important, BSD relies on paging.

All operand references on the 386 are tied to one of the segment registers. This segment register uses a 16-bit selec-

386 Segmentation and Paging

The 386 has six segment registers (CS, DS, SS, ES, FS, and GS) which can select one of 16,383 (8,191 shared and 8,192 private) segment descriptors. These segment descriptors reside in either the Global Descriptor Table (GDT) or the Local Descriptor Table (LDT) and determine underlying characteristics (type attributes, location in linear address space, and segment size). In addition to memory segments, system segments are available to the operating system for special purposes and call gates to facilitate controlled indirection into other possibly hidden segments.

Memory segments can be selected via a dedicated segment register, with different results. The CS register contains program instructions. The DS register selects program data. The SS register selects the program stack. The ES register selects the destination of string instructions. Both the FS and GS registers are undedicated at this time. It is even possible to reassign the segment registers in the machine instructions, so one can view the ES, FS, and GS segment registers as alternative DS segment registers.

Each memory segment has a size, and can be as large as 4 gigabytes. In order for that segment to be active, however, it must consume space (global linear address space) in direct proportion to its size. This means that, although a process may possess a total address space greater than 4 gigabytes, only an *aggregate* of active segments totaling less than or equal to 4 gigabytes is permitted. While the 386 theoretically can address $2^{14} \times 2^{46}$ bytes, *in practice* only 2^{32} bytes (4 gigabytes) can be active at any time. If the maximum 4 gigabytes of instruction, data, and stack (for both operating system and each user process) is invoked, managing the global linear address space to allow segments to be active (present) when linear address space is available becomes a significant problem.

Segments can also be overlapped in linear address space. Because through both segments we can access the same memory interchangeably, possibly with different attributes, this overlap is called an *alias*.

80*x*86 segments can be either "bottom up" or "top down." A segment that is bottom up means that one begins with segment relative address 0 and "grows up" to the desired address x (that is, [0 . . . x]). A segment that is top down means that one begins with segment relative address 0xffffffff and "grows down" to the desired address y ([y . . . 0xffffffff]). (Yes, we know this is awkward, but that's how it works). Segments are grown only in accordance to these rules. The stack segment is the only common example of a downward growing segment.

Many other attributes are provided that control the type of access allowed within the segment. The designers of the 386 prefer segments be used in memory protection regulation, and have provided a plethora of features not found in the paging unit. Segment attributes, such as 32-bit vs. 16-bit operations, byte vs. page granularity, and user vs. supervisor mode, control the mode of the microprocessor, depending on the segments that are actually in use.

It is quite costly to implement segments in the microprocessor. That is why underlying shadow registers, invisible to the programmer, are used. They provide a hardware "assist" to the segmentation functionality.

We manage to avoid many paging bookkeeping problems by running in "flat" mode. This is accomplished by aliasing the CS, DS, SS, and ES segment registers to the exact same linear address space (see Figure 4), thus making it an identity function. We can then regard any of the intrasegment addresses as if they were *linear address space*. Of course, this ends up defeating the advantages

tor (low-order bits determine level of access) to find a descriptor. This descriptor then determines the location of underlying memory in *linear address space*. When segmentation alone is enabled (also known as protected mode), the linear address space corresponds to the physical address of the selected segment for the operand. However, when paging is implemented, the *linear address space* address must be run through a two-level paging mechanism to find the *physical page frame number*, the actual address of physical memory underneath the virtual address.

One of the most powerful, yet confusing, features of the 386 is its segmented architecture. While the current trend in microprocessors has been oriented towards a single "flat" linear virtual address space, the 386 has continued the bias toward segments held by the entire 80*x*86 line. The two most important changes in the 386 from previous versions — permitting 32-bit operations and expanding segments from 64 Kbytes to 4 gigabytes in size — may turn some of the inherent disadvantages of 80*x*86 segments into an advantage. Segments once too small for many data items (such as arrays of real numbers) can now utilize alternative address spaces. This is of great interest to those working with specialized applications, such as 3-D to 2-D transformations.

Segmentation and 386BSD

UNIX was initially developed on machines that relied on linear virtual address spaces. As such, Berkeley UNIX provides no support for segments and instead expects a large linear virtual address space for both kernel and user. In fact, UNIX in general adapts to segments only under duress.

Originally, we had intended to use segments in a straightforward manner. However, we found that would result in a host of nuisance problems. For example, many programs (debuggers, assemblers, and object-linking editors) must be modified so that separate address spaces for the various regions could be maintained. Object file format, always in a state of flux due to the varying degrees of dynamic loading of instruction and data structures, would *require* change.

Another problem which arises when using segments is that the shared data in the instruction segment requires strict typing in the assembler (we force instructions to reference the CS segment directly) to obtain access. Because some compilers put data constants in the code area with the intent of sharing memory used by other processes, invoking segments would create little problems everywhere for the compiler.

Still other problems result from the use of string instructions on stack resident data and that time honored bad practice known as self-modifying code. The key flaw in all these cases is that the binding to the particular segment register is mandated by the assembler, and cannot be properly resolved by the object code linker as other symbols are

---

of segments as well.

Some new microprocessors, such as the 386, feature architectures which exploit large segments. This is because 4 gigabytes is starting to fill up, and going to 64-bit addresses will not be happening soon. Many would argue that 4 gigabytes will never be filled, but history states otherwise. 64-Mbit RAM is already on the drawing boards — in fact, some actually exist. In a few years, it will be commercially available. Because a typical computer uses on average 64 to 128 RAM chips, with many companies currently offering 64-Mbyte systems (512 1-Mbit RAM), it will not be long before a computer with 512 64-Mbit RAM chips (4 gigabytes) is introduced. As such, segmented architectures may provide a way of spanning the address space gap that could result.

It's amazing that at the beginning of the microcomputer revolution, an Altair 8800 with 4 Kbyte of RAM was considered incredible because it could run Basic! How times change.

We have seen how segmentation works in the 386. Now let's examine paging. For our purposes, segmentation on the 386 is defeated by running in "flat" mode. We can then consider intrasegment addresses as if they are *linear address space*.

Paging works with a two-level scheme that permits the sparse allocation of address space, so that the whole address space, or even all of the address space mapping information, need not be present. Otherwise, a 4 gigabyte process would require more than 4 Mbyte of page tables, even though it may be the case that only a few thousand would be active at any time. Typically, for our purposes, only three pages of page tables are allocated per process (page directory and the top and bottom address space page tables). This is sufficient to run a 4-Mbyte process (instruction plus data size) and 4 Mbyte of stack. (Note that all processes run with a full-sized address space and can dynamically grow to use it.) This mechanism is quite successful in reducing memory-management overhead.

The two-level scheme splits the incoming virtual address into three parts: 10 bits of page table directory index, 10 bits of page table index, and 12 bits of offset within a page. The page table directory is a single page of physical memory that facilitates allocation of page table space by breaking it up into 4-Mbyte chunks of linear address space per each of its 1024 PDEs (Page Directory Entry), which determine the location of underlying page tables in physical memory.

Each PDE-addressed page of a page table contains 1024 PTEs (Page Table Entry). A PTE is similar in form and function to a PDE. The major difference between a PDE and a PTE is that a PTE selects the physical page frame for the desired reference. Once the frame offset least-significant address bits are obtained, the final address is determined. This method is identical to that used in many other common microprocessors (the MC68030, Clipper, and NS32532, among others).

Each PDE and PTE may be marked either "invalid" (not currently used) or "valid" (the underlying page of physical memory is present). In addition, other attribute bits mark entries as "read only" or "read-write" and "supervisor" or "user." Because segmentation is not used to control memory protection, we keep processes honest by relying entirely on the paging mechanism's attributes for protection as well as for the allocation of memory.

The mechanism to convert virtual to physical addresses is quite elaborate. To speed things up, the 386 keeps a Translation Look-aside Buffer (TLB) of 64 cached entries, managed entirely transparently. One side affect of this hardware is that if the operating system changes any of the page tables that may be in use, it must flush this cache. The 386 does not allow selective flushing — only a complete flush of all cache entries by reloading the page directory address register *cr3*. This is an expensive operation which may be repeatedly performed as we successively transform an address mapping of a process within the kernel (as many as six times in the worst case).

—B.J., L.J.

normally handled.

Given all of the problems which arose and, in accordance with our 386BSD goals, we chose to minimize support for segmentation by running the machine in "flat" mode. As a result, no tinkering with object file format or tools was required. An amusing side effect of this approach is that it allowed us to cross-develop 386 code on VAX and NSC32000-based computers using the native object utilities. This choice minimized bookkeeping considerably but also ultimately defeated the purpose of segments. A more elaborate design was beyond the scope of our project.

Kernel Linear Address Space Overhead

The kernel, as well as the user mode programs, requires its own set of segment registers. If the kernel is called, its segments must be present. This takes up precious linear address space. Thus, we can never run a process exactly 4 gigabytes in size because a portion of the address space must be reserved for kernel use. Even if we try to use segments to relocate the kernel, we cannot escape the limit — it not only takes up the same linear address space but also forces us to use intersegmental instructions to communicate data between user process and the kernel. Since the user, the process, and the kernel must share virtual address space, we limited ourselves to a maximum process size of 4 gigabytes less the kernel size.

The kernel segment registers are outlined in Figure 4. These segment registers cover (alias) the user segments and allow access to the user space from the kernel in any way desired (read, write, or execute). Because all segments start at zero, the kernel begins at a high address (or offset) and always runs relocated. In 386BSD, the code segment just covers the end of the kernel instruction region, because no self-modifying code was needed.

One way to avoid linear address space sharing constraints is to have all interrupts, traps, exceptions, and system calls internally context switch to a separate process to execute UNIX system functions, using the 386 trap with task switching feature. This unique 386 hardware allows traps to be handled by either procedures or tasks. However, task switching is very expensive and the system would context switch thousands of times more frequently than otherwise. Also,

(continued on page 28)

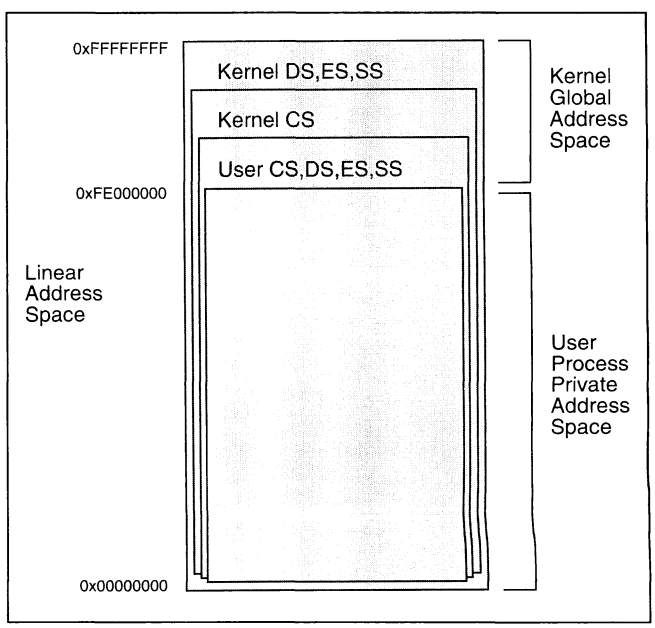


Figure 4: *386BSD segment registers*

Zortech C++

MS-DOS • WINDOWS • OS/2 • DOS 386 • UNIX 386

applications in this exciting new environment. Do you need MS-Windows class libraries? Call for details of third party Zortech Validated Products.

OS/2 NEW

The OS/2 Developer's Edition now provides a C++ Compiler and source level Debugger designed for C++. In the words of OS/2 Magazine:

"Zortech C++ serves as a direct replacement for the Microsoft C Compiler in developing applications, allowing programmers to use object-oriented techniques in OS/2 development."

The OS/2 Developer's Edition also includes C++ Tools, Flash Graphics and C++ Workbench for OS/2 together with the standard DOS Developer's Edition.

Upgrades for existing OS/2 Compiler Option owners now available. Please call for details.

UNIX 386 NEW

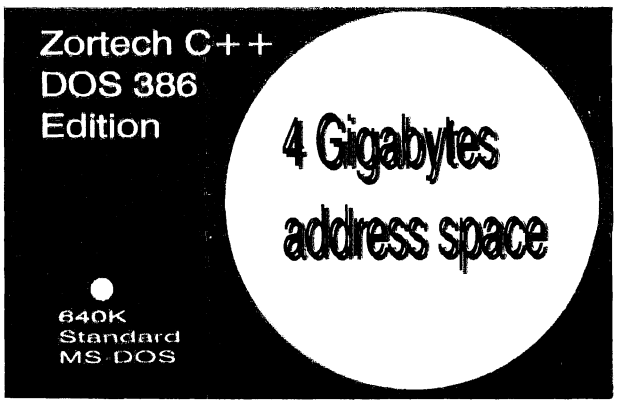
Not a day passes at Zortech HQ without numerous requests for a UNIX version of Zortech C++. Now, DOS and OS/2 developer's can reach new markets by easily moving their code to SCO UNIX 386 and binary compatibles.

The Zortech C++ V2.1 UNIX 386 Compiler generates the same tight, fast code that Zortech's DOS and OS/2 users have come to expect. UNIX specific versions of Flash Graphics and the C++ Workbench are also provided.

In line with the traditional Zortech Policy, owners of the Zortech C++ V2.1 UNIX 386 Compiler will be able to inexpensively upgrade to the forthcoming Zortech C++ V2.1 UNIX 386 Developer's Edition.

DOS 386 NEW

Now, with the 386 you can address up to 4 Gigabytes of memory. Why spend so much money on 386 hardware and not use software which will take advantage of it?



On the other hand, you need to retain the facilities of standard MS-DOS too.

MS-DOS developers can now build true 32 bit C and C++ applications for 386 processors using Zortech's powerful 386 development system. The Zortech C++ V2.1 Developer's Edition for DOS 386, contains 32 bit versions of the C and C++ Compiler, Flash Graphics library, C++ Debugger and full standard library source code together with all the familiar features provided with the standard DOS Developer's Edition.

Using Phar Lapp's much acclaimed 386/DOS Extender Technology, you can build applications which access 4 Gigabytes of linearly addressable memory. Your applications will also be Plug & Go for use with Phar Lapp's 386 DOS Extender which may be purchased seperately.

C++ VIDEO COURSE

Zortech's C++ Video Course is all the training material you need to move a team of good C programmers into the world of C++. Many corporations have already done just this.

Cut the hotel bills, travel expenses and fees of outside training courses and seminars - not to mention the inconvenience and disruption to your normal routine.

Use a proven training tool, that in one hour a day, over a period of six weeks, can train your whole team in C++ for the price of one airline ticket.

The course consists of 32 tutorials on six one hour VHS tapes *together* with one 256 page workbook containing course notes and exercises. Unlimited additional course workbooks are available at modest cost. Compiler & hardware independent. NTSC or PAL format available.

C++ FOR MACINTOSH CALL FOR DETAILS

(continued from page 24)
the UNIX kernel is not intended to run itself as a process, as use of this feature would require.

Virtual Address Space Layout

Within the 4 gigabytes per process address space, a process must be allocated regions for instruction, data, and stack for both user programs and the kernel. Some of these regions

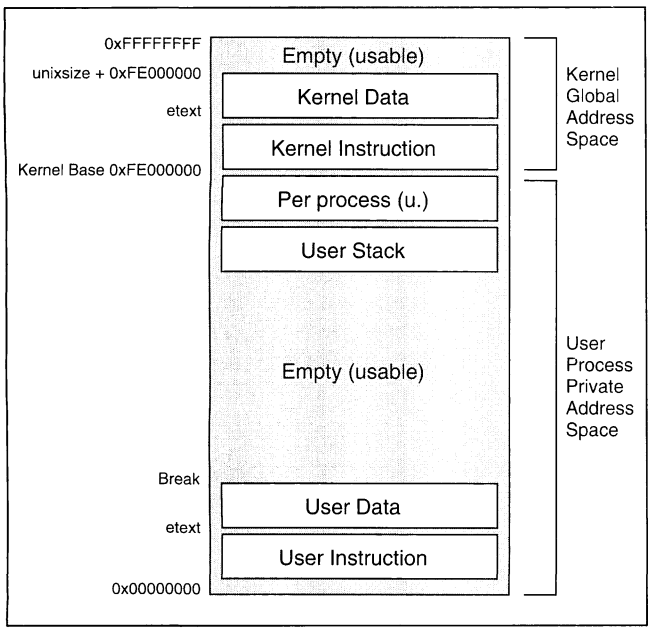


Figure 5: *386BSD process virtual address space*

(user data, user stack) must grow as a process runs, and support must be available for additional regions used for shared memory and shared/dynamically loaded libraries. The size of these regions and their placement becomes an important consideration for any UNIX port.

The traditional UNIX approach is to place the instruction region at the beginning of the address space, followed by data, unused space, and finally a stack region. The purpose of the empty space is to build in room so that the stack can grow down and the data (for heap storage) can grow up. The end-point is known in UNIX vernacular as the "break." Usually, text starts at absolute virtual address 0.

A problem common with UNIX systems arose from the extensive use of uninitialized string pointers, which by default were set to the value 0. Because the first word at address 0 was also set to 0, this meant that null pointers always pointed to null strings. However, many early computers did not permit the bottom of address space to be used in this way and a tested program would abort. UNIX code that was thought "proven" on the PDP-11 and VAX was actually masked by the development system architecture. Eventually, many uninitialized pointers were located and corrected. Some versions of UNIX also leave the very bottom and top of address space unmapped to catch in directions through 0 and -1. This method is of limited effectiveness, however, if a structure referenced through such a pointer is bigger than the size of the bottom and top address space holes.

386BSD virtual address space is arranged in the traditional manner (see Figure 5). The user address space begins at zero with text, (yes, we do indeed have 0 at location 0), followed by data, unused space, and finally the stack. The start of the user stack, located at the top of the user's address space, is not fixed. (A future project may utilize this feature to "lower" the stack, providing room for dynamically created regions.) Because only the operating system needs to know the exact location of the user stack, it assigns the stack's address space on process program load (*exec* system call).

Per-Process Data Structures

The kernel address space resides above the user portion of the process virtual address space. By virtue of being co-resident in the virtual address space with the user space (a somewhat mandatory virtue), the kernel can directly reference any part of the current running user process in the lower portion of memory.

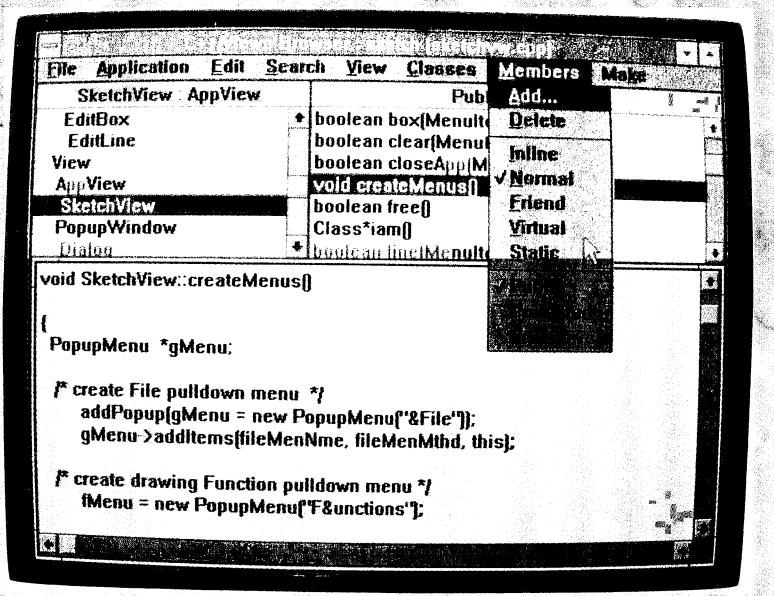
As in the user space (and in UNIX executable files), kernel instructions and data are arranged consecutively. The stack and a new special region, the *per-process data structure* or *user structure* (*u.* for short), appear below the kernel. One advantage of this arrangement is that it becomes possible to share all portions of the page tables for address space above the kernel base address. Notice that though this is a vital part of the kernel, it is technically at the very top of user address space and is purposely left readable by the user process. Everything beneath the system base address is switched when a context switch to the next process occurs.

Currently, the kernel address space starts at virtual address 0xfe000000, and allows up to 32 Mbyte of address to be reserved for use within the kernel. This boundary can be moved at a later date if more address space is needed.

Access to the ISA bus device memory (screen and LAN buffers) is obtained through an allocated region of the kernel memory, known as a utility page map. This is similar to portions of on-demand physical memory used by the kernel through other utility page maps. The kernel also has a variety of data structures scaled and allocated at boot time (*valloc*) and a heap for dynamic demands (*malloc*).

C++/Views™

for Microsoft Windows



THE MICROSOFT WINDOWS 3.0 DEVELOPMENT TOOL THAT DELIVERS FROM START TO FINISH.

C++/Views is a development tool for C++ programmers that not only reduces the complexity of Microsoft Windows **3.0** but also slashes development time by up to 75%.

Delivers on the promise of Object-Oriented Programming (OOP).
Encapsulates more MS Windows 3.0 functionality than any other tool on the market today. Get MS Windows applications off to a fast-start with a framework of over 65 tested and ready-to-go C++ classes.

Has the most complete C++ class library for MS Windows Development.
Get started with graphical user interface classes such as windows, views, bitmaps, dialog boxes, menus, popup menus, graphics, regions, pens, brushes, controls, buttons, listboxes, valuators, editors, printers and much more. Organize your data with foundation classes such as containers, collections, sets, dictionaries, files, strings, streams and so-on. Use other classes to manage the persistance of objects across files, to perform serial communications, and to activate timed events.

Provides support for the entire project.
Comes with a complete OOP development environment including the first fully functional C++ class hierarchy *Browser*. Also includes an *Interface Generator* for building C++ dialog classes and a *Documentor* for automatically producing high quality documentation of your classes.

Works with Zortech C++.
Combine C++/Views with the **Zortech C++** compiler for a cost-effective and highly productive development environment for building your next generation software systems.

Pays for itself on even the smallest project.
Only **$495.00** with no royalties.
Comes complete with source code.

C++/Views
from CNS, Inc.

CNS, Inc., Software Products Dept.
7090 Shady Oak Road, Minneapolis, MN 55344
(612)-944-0170 • FAX (612)-944-0923

MICROSOFT WINDOWS
Version 3.0 Compatible Product

CIRCLE NO. 457 ON READER SERVICE CARD

(continued from page 28)

386 Virtual Memory Address Translation Mechanism

The 386 paging mechanism impacts the 386BSD specification with respect to address space allocation constants: Each page is 4K byte in size and must reflect the minimum granularity of address space allocation, while each page of page tables maps 4 Mbyte of address space. These constants determine address boundaries used to allocate memory and share address space between similar processes. Shared objects starting on 4-Mbyte boundaries can share page tables as well as underlying physical memory.

Page size granularity is important to the layout of executable files. Instruction and data regions are arranged into discrete and aligned memory page units, so that it is possible to demand load pages that may be either "read-only" (instructions) or "read-write" (data or stack). The page table size granularity is typically located at the beginning of each user, user stack, and kernel address space. It is possible to share these among many processes, obviating the need for separate page tables. As a result, while each process has its own page table directory, the top eight PDEs of each process page table directory point to the same kernel page tables. Thus, the kernel's portion of address space is global to all other processes.

User to Kernel Communication Primitives

By arranging our address space as outlined, we've greatly simplified the routines that communicate between kernel and user process (now the kernel routines can directly access user space). All that is needed is a way to determine if a selected portion of user memory may be read or written before it is attempted. On some machines (such as the VAX) special instructions are available for this purpose. The 386, however, offers instructions only for use in validating segments, not pages. So we must use a different strategy.

In 386BSD, we chose to set a global variable (*nofault*) to a nonzero value. If a fault happens during any user/kernel communication primitive, it transfers to the address held within no fault. In this way we can catch illegal references by using the microprocessor's own address translation mechanism to find them, instead of by tedious code evaluation on every reference.

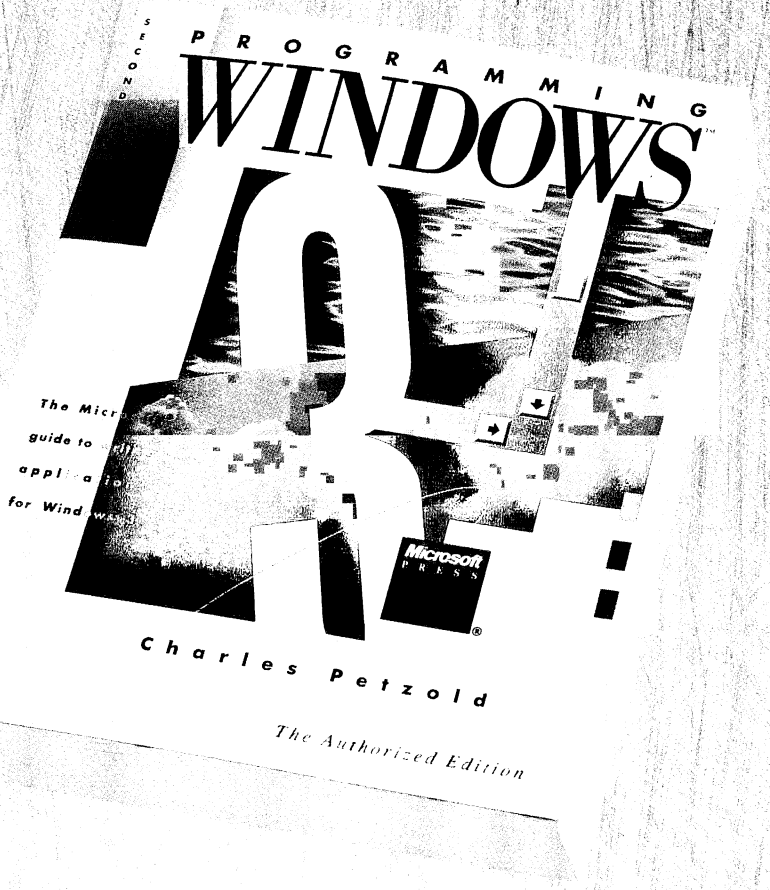
Unfortunately, one idiosyncrasy of the 386 now rears its ugly head. The designers of the 386 decided that segment attributes should be used to ultimately determine access to regions in a process, thus making their use mandatory in the system even if we don't need them. To be precise, we have page attribute bits that can be used for protection. These work as expected, unless the 386 is run in supervisor mode (as does the kernel). In this case, only the valid/invalid attribute has any effect. This nuisance or "feature" requires a bit of workaround to make the primitives complete.

Berkeley UNIX Virtual Memory System Strategy

The current Berkeley UNIX virtual memory management subsystem was originally designed for use with a VAX, and as such has no support for page directories. For that matter, the 386 doesn't know of such VAX concepts as P0 and P1 address spaces for instruction/data and stack nor of page table-length registers. Currently, these are simulated in 386BSD. However, work is underway to revise the entire virtual memory system to permit more generalized operation over all supported Berkeley UNIX platforms, now that the demands of each platform have been made obvious.

Portions of the VAX were simulated by employing code, written by Mike Hibler at the University of Utah, which supports the 68030 paging memory management. Because the 386 code is so similar, we used a conditional compila-

Petzold on Windows™ 3



The classic guide to programming for Microsoft® Windows is now updated to address Microsoft Windows version 3.0. Whether you're looking at Windows programming for the first time or converting your existing applications to take advantage of Windows 3, Charles Petzold provides the overview information and hands-on detail that you need. Accept no substitutes. Ask for PROGRAMMING WINDOWS, Second Edition, by Charles Petzold at your local bookstore or order directly from Microsoft Press. **$29.95.**

To place your credit card order, call
1-800-MSPRESS.
Please refer to campaign AH.
Microsoft Press
One Microsoft Way
Redmond, WA 98052-6399

Microsoft PRESS®

An Authorized Edition

CIRCLE NO. 165 ON READER SERVICE CARD

(continued from page 30)
tion that shares 68030 and 386 versions interchangeably — an odd couple indeed.

Structure of Per-Process Data (u.)

Within each process accessed by the kernel exists a unique data structure containing the private variables of the process used to provide UNIX system call functionality. This is called the "extended state" of a given process and is collected into one location. If the process is long inactive, this state is swapped to secondary storage to reclaim RAM memory. All of the machine-dependent fields in this structure lie within the first element *u_pcb*, a process context descriptor. However, the size of this structure and its adjoining kernel stack is also a machine-dependent parameter. The *u.* is currently defined at about 1 Kbyte in size. This fits amply within a single page.

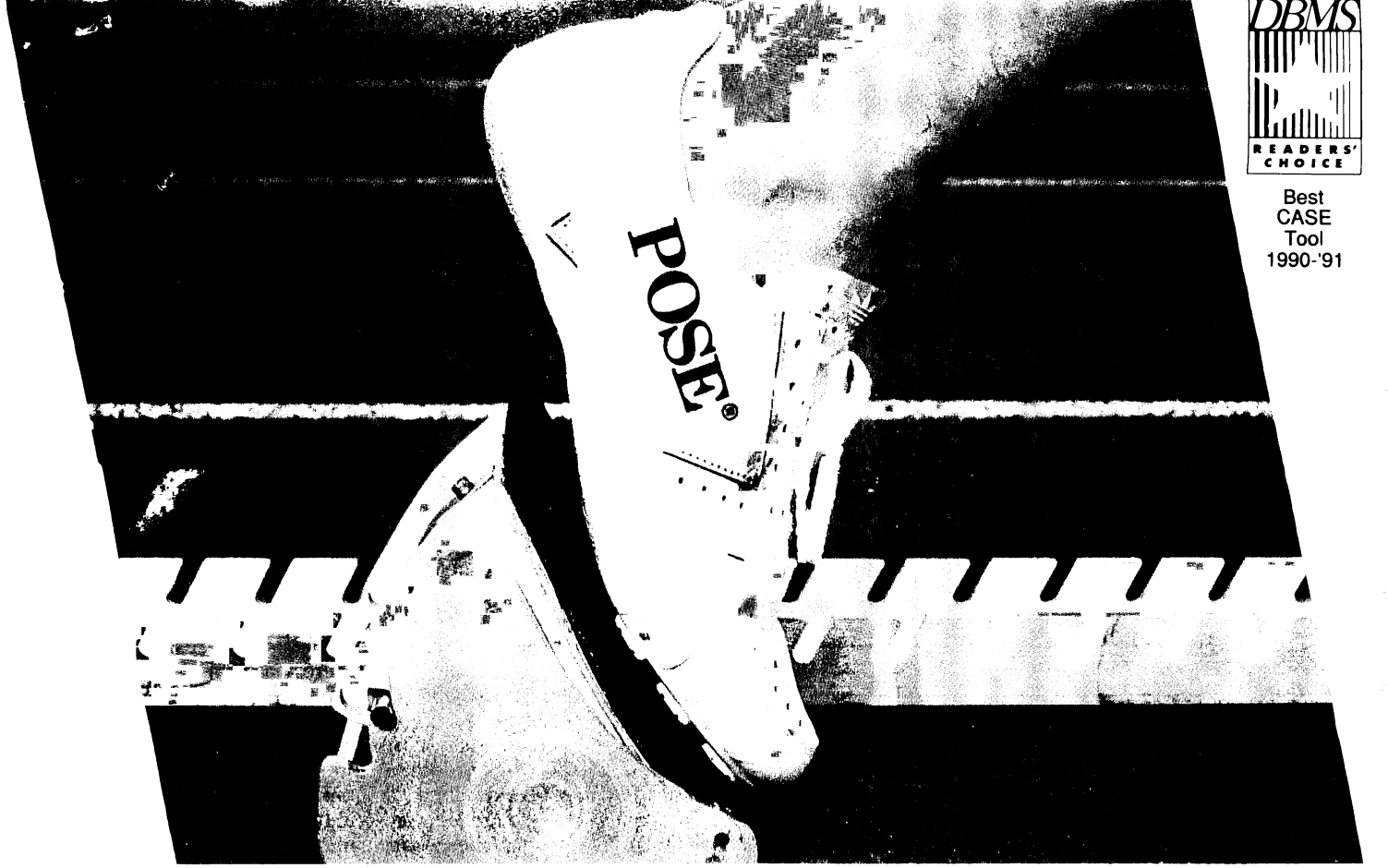
Another page is sufficient to hold a kernel stack. This results in a per-process data structure two pages in size. By leaving these as two separate pages in 386BSD, instead of combining them into a single page (giving us a smaller kernel stack), the kernel stack segment can be used to catch the stack overflow ("redstack") condition. This will appear as a future enhancement.

Process Context Description

As seen in Figure 6, the process control block (*struct pcb*), contains the 386-specific per-process information. This is

```
/* Intel 386 process control block  */

struct pcb {
        struct  i386tss pcbtss;
#define  pcb_ksp  pcbtss.tss_esp0
#define  pcb_ptd  pcbtss.tss_cr3
#define  pcb_pc   pcbtss.tss_eip
#define  pcb_psl  pcbtss.tss_eflags
#define  pcb_usp  pcbtss.tss_esp
#define  pcb_fp   pcbtss.tss_ebp

/* Software pcb (extension)  */
        int     pcb_fpsav;
#define  FP_NEEDSAVE        0x1     /* need save on next context switch */
#define  FP_NEEDRESTORE     0x2     /* need restore on next DNA fault */
        struct  save87  pcb_savefpu;
        struct  pte     *pcb_p0br;
        struct  pte     *pcb_p1br;
        int     pcb_p0lr;
        int     pcb_p1lr;
        int     pcb_szpt;           /* number of pages of user page table */
        int     pcb_cmap2;
        int     *pcb_sswap;
        long    pcb_sigc[8];        /* sigcode actually 19 bytes */
        int     pcb_iml;            /* interrupt mask level */
};

/* Intel 386 Task Switch State  */
struct i386tss {
        long    tss_link;   /* actually 16 bits: top 16 bits must be zero */
        long    tss_esp0;   /* kernel stack pointer priviledge level 0 */
#define  tss_ksp  tss_esp0
        long    tss_ss0;    /* actually 16 bits: top 16 bits must be zero */
        long    tss_esp1;   /* kernel stack pointer priviledge level 1 */
        long    tss_ss1;    /* actually 16 bits: top 16 bits must be zero */
        long    tss_esp2;   /* kernel stack pointer priviledge level 2 */
        long    tss_ss2;    /* actually 16 bits: top 16 bits must be zero */
        long    tss_cr3;    /* page table directory physical address */
#define  tss_ptd  tss_cr3
        long    tss_eip;    /* program counter */
#define  tss_pc   tss_eip
        long    tss_eflags; /* program status longword */
#define  tss_psl  tss_eflags
        long    tss_eax;
        long    tss_ecx;
        long    tss_edx;
        long    tss_ebx;
        long    tss_esp;    /* user stack pointer */
#define  tss_usp  tss_esp
        long    tss_ebp;    /* user frame pointer */
#define  tss_fp   tss_ebp
        long    tss_esi;
        long    tss_edi;
        long    tss_es;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_cs;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_ss;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_ds;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_fs;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_gs;     /* actually 16 bits: top 16 bits must be zero */
        long    tss_ldt;    /* actually 16 bits: top 16 bits must be zero */
        long    tss_ioopt;  /* options & io offset bitmap: currently zero */
                            /* XXX unimplemented .. i/o permission bitmap */
};
```

Figure 6: *Process control block*



DBMS READERS' CHOICE

Best CASE Tool 1990-'91

Start With Results. Win The CASE Race.

The CASE race is on, and victory goes to swift results. A CASE product must not be more difficult to use than the problems it solves. In today's development environment nobody has the time to learn CASE. With POSE you purchase and learn modules in the same sequence as solving your system design problem, so you learn and produce results simultaneously.

Start with as much or as little as you need. The POSE DOS-based product line will be with you all the way from entry level explorations to full **AD/Cycle** compliance.

READY to solve problems.

Our modular design allows you to buy and use CASE as you need it. You might start with Data Modeling, then add Process Modeling, Screen Prototyping, Planning Matrix, and finally Cobol code generation. POSE generates DB2, SQL/DS, ORACLE, FOCUS, AS/400, and ADABAS schemas.

SET the pace for fast action.

Thorough documentation and a graphical user interface make intuitive CASE a reality. Choose from Yourdon, Gane/Sarson, or a variety of other methodologies. There is no better way to learn CASE than to use CASE.

Deciding that you need CASE is easy. Getting it into your organization without causing a revolution is something different. POSE's low cost and broad product line make it a natural for convincing skeptics. Our 30-day money-back guarantee insures a risk-free decision.

GO for the three E's.

Easy to learn. Easy to use. Easy to justify. Find out how POSE can make the three E's work for you. Call today and ask about free demo's and our risk-free money-back guarantee.

1-800-537-4262

ADJ11

Call Today

1-800-537-4262

CSA

COMPUTER SYSTEMS ADVISERS, INC.
50 Tice Boulevard
Woodcliff Lake, NJ 07675
(201) 391-6500 (800) 537-4262
Telefax: (201) 391-2210

CIRCLE NO. 413 ON READER SERVICE CARD

(continued from page 32)
broken down into hardware-dependent fields and software-related fields. The process control block is placed at the front of the user structure so that the information can be reloaded from the address of the user structure and force

It is quite costly to implement segments in the microprocessor. That is why underlying shadow registers, invisible to the programmer, are used. They provide a hardware "assist" to the segmentation functionality

active a previously inactive process. The user structure address is recorded in the process table. Each entry describes global information about a process.

The 386's hardware context switch facility can be used to switch from process to process. By placing the hardware-dependent information at the beginning of the process control block, in the form of the 386's Task Switch State (TSS) data structure, it is possible to switch from one process to another with a single intersegment *ljmp* instruction to the appropriate task gate selector. While this feature has been implemented in 386BSD, it is not used at this time for switching between processes due to performance considerations. However, it can be used in other cases, such as exception handling, and we may elect to use it for process switching in the future. We view this as one of those rare "have your cake and eat it too" decisions.

In 386BSD, not all hardware context is switched in this manner, because some processes never access the large amount of state information (108 bytes) used by the numeric coprocessor. We allow for this with the *pcb_fpusav* structure. Other fields correspond to some implementation demands specific to Berkeley UNIX, including simulating VAX hardware constructs invoked by the virtual memory system not existing on the 386. Fortunately, this was a small amount of code. It is a tribute to the concept of UNIX that the machine-dependent portion of the system is as small as it is.

Page Fault and Segmentation Fault Mechanism

To report exceptions that occur in the 386 memory management hardware, they must be caught and routed to the proper portion of the kernel. UNIX places these exceptions in two categories: Faults signaled to the user process, which terminates the process if it is not interested in the exception, and "resource not present" faults sent to the virtual memory system to request a missing page.

The 386 also signals a variety of segment exceptions, almost all of which result in dire consequences for the process that invokes them. A single page fault exception encodes both "page not present" as well as "protection violation" events. These page faults, along with the fault address, are recorded in processor special register *cr2* and should be carefully examined to determine the precise nature of each exception.

Death Taxes Software Piracy

We can save you from one of them.

Sorry. Death we can't do anything about. As for taxes, when you use our product you'll probably wind up paying more. But software piracy: there we offer some help. Our family of software protection devices (dongles) have improved unit sales for over 2,000 companies around the world. Our products can be used in the MS-DOS, OS/2 and Macintosh environments.

Build Your Own Custom Protection Environment

Use our patented "dual-locking" ASIC chip as the basic building platform. Next, add options like: on-the-fly read/write memory, write-once or multiple-write locking codes, and encryption shells. Then add your own programming creativity to build a protection environment best suited to your product.

Users attach the device to their parallel port, and programs won't run without it. Back-up copies, hard disk and LAN operation are not interfered with.

Your Intellectual Property Belongs To You

And if you don't protect it, who will? Our products offer the most equitable way to protect your interests without sacrificing the rights of your customers. Call us today for information and demonstration units.

Software Security

1011 High Ridge Road - Stamford, CT 06905
1-800-333-0407 ext. 102
203-329-8870 Fax 203-329-7428 BBS 203-329-7253
AppleLink™ D2379

(continued from page 34)

Other Processor Faults

Along with address space faults, we found we must map 15 other faults (see Figure 7) into the Berkeley UNIX kernel exception-handling mechanisms. The numeric coprocessor presents special fault-handling challenges, for it can be operating when 386BSD switches to another unrelated process. In that case, we can get a trap that should have been passed to a process other than the one currently running.

If 386BSD receives an unexpected fault while running in the kernel, it must immediately force the kernel down (in UNIX vernacular, to "panic") and attempt to save as much state information as possible for diagnostic purposes. Thus, we differentiated user traps from kernel traps. In most other microprocessors, a bit in the processor flags or status word determines if we are running in the kernel, but the 386 offers no such bit. So, 386BSD examines the contents of the CS segment register when a trap occurs (this is saved by the hardware during an exception) to determine if an instruction was executing in user mode.

We found a hornet's nest of microprocessor idiosyncrasies unique to a 386 UNIX port

Microprocessor Idiosyncrasies

We found a hornet's nest of microprocessor idiosyncrasies unique to a 386 UNIX port. Some of the primary issues these touched upon included that of switching from real mode (20-bit addressing) to protected mode (32-bit addressing), creating segment descriptors to fill the interrupt descriptor table, creating other segments for use by the user and kernel modes of a process, and finally, novel surprises between different steppings of the 386/486 themselves.

One major irritant was the need for at least one TSS structure to be present at any time, even if we didn't use a TSS for task switching. The TSS records the contents of the kernel's stack pointer for use when the kernel is reentered from user mode (interrupt, exception, and system call). Our early versions of 386BSD worked well as it started up within the kernel, moved into user mode for the first process, and then froze after hitting the first system call. Imagine our surprise when we found that, in effect, it had no place to save where it was coming from on the kernel stack!

System Call Interface

A table of system calls is provided by Berkeley UNIX with the assigned index number that differentiates them. This table specifies, in part, a binary standard for system calls — in this case, of a POSIX-based system. Of course, because POSIX is considered an "object library" definition (as opposed to the regulation at the system call level desired by ABI and BCS advocates), one might accurately consider this an "academic" standard. In deference to these other standards, however, we chose to accept their suggested format for system calls.

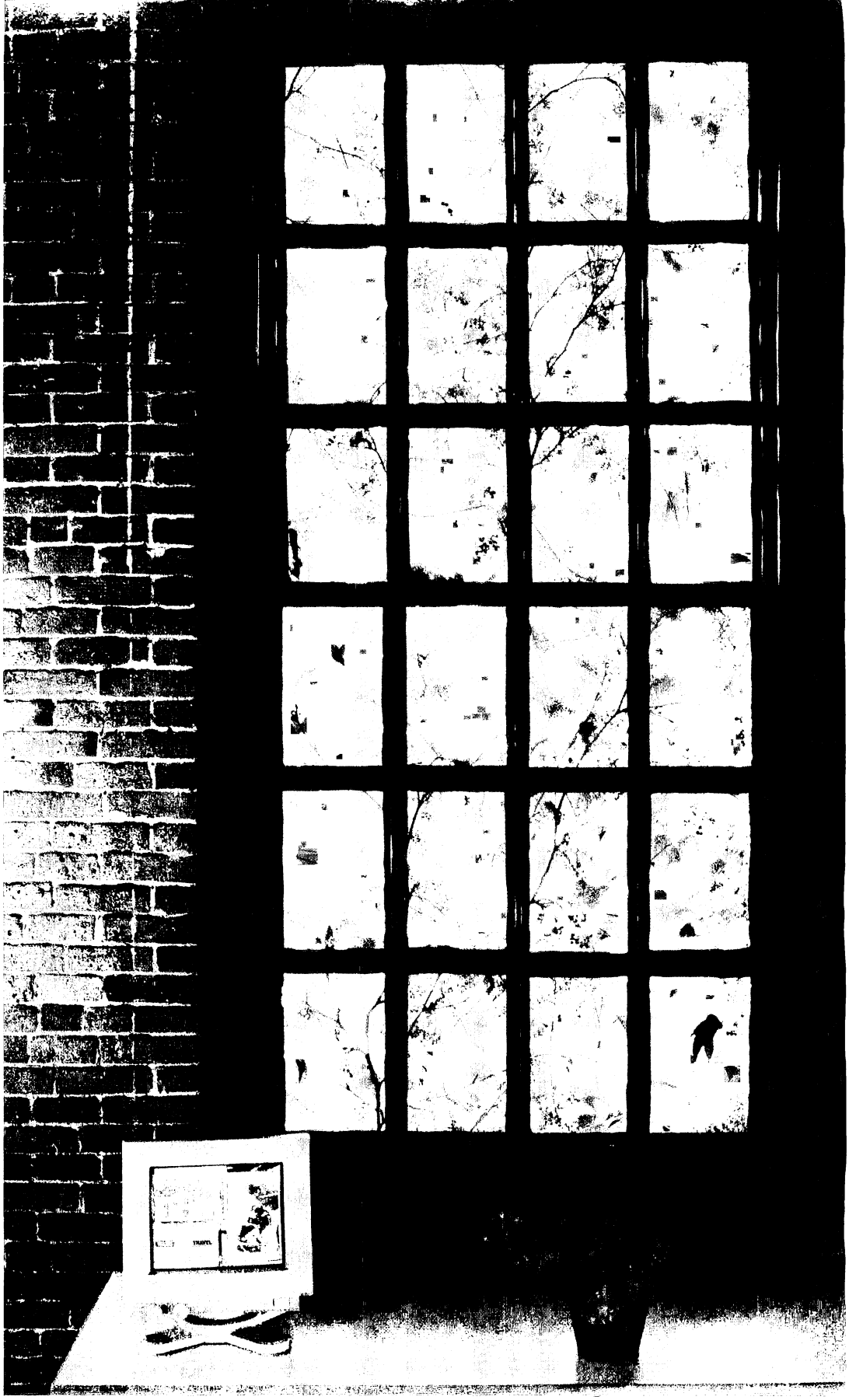
Figure 8 is a code template for the system call stub used in 386BSD, in this case a write system call. The *lcall* instruc-

Fill your windows with images...

Despite the billions of dollars spent on computers, networks and software to automate business functions, studies have shown that only 5% of all the information needed is available online.

Wang OPEN/image Developer Kits give you tools to easily add imaging to your Microsoft® Window applications. Wang OPEN/image Windows are invoked within MS Windows as a set of Dynamic Link Libraries (DLLs) allowing your applications to capture, scan, view, edit, index, store, print or distribute images without requiring any special hardware.

For more information about Wang's OPEN/image Developers Kit, call 800-TEL-WANG.



WANG

Microsoft is a registered trademark of the Microsoft Corporation.

OPEN/image Windows

CIRCLE NO. 531 ON READER SERVICE CARD

(continued from page 36)
tion is an intersegmental call instruction that references a special segment selector, known to be a UNIX system call gate into the kernel. The selector corresponds to the first descriptor in the processes local descriptor table. To designate which system call is to be used, the *eax* register is loaded with the index from the table. Arguments for each system call are present on the stack, and this stub is called from another procedure. System calls return after the *lcall* instruction, returning values in the *eax* and *edx* registers (just as other C procedures do). System calls report failure by setting the carry bit and recording error notification in *eax*.

System Specific (ISA) Issues

So far, we have only described issues relating to our choice of microprocessor. But this specification is incomplete unless the issues relating to the bus and the system surrounding the microprocessor are examined. We recognized that the 386 already operates on a plethora of different buses, including ISA, EISA, MCA, VME, and MULTIBUS, and that these issues vary depending on which bus is used. We may even need to support more than one bus at a time, or even a custom bus. As such, we decided that 386BSD must take into consideration the support requirements of many different bus standards.

386 Processor Exceptions

Exception	Description	Pushes an Error Code?
Divide	Divison by 0 or division overflow	No
Debug/Trace	Single step or debug hardware condition	No
Breakpoint	Executed an INT3 instruction	No
Overflow	Executed an INTO instruction when OF bit set	No
Bounds Check	Executed an BOUND instruction which failed	No
Illegal Instruction	Executed an unknown instruction	No
NPX DNA	Numeric processor device not available	No
Double Fault	Recursive fault (fault while processing a fault)	Yes
NPX Operand	Numeric processor accessed outside of segment	No
Invalid TSS	Attempted to task switch to incorrect task state	Yes
Segment Not Present	Attempt to access a not present descriptor	Yes
Stack Segment	Problem with current stack descriptor	Yes
General Protection	Protection problem with a segment descriptor	Yes
Page	Page missing or protection problem with address	Yes
NPX Error	Numeric processor signals an error	No

Figure 7: *386 processor exceptions that needed to be mapped into the kernel exception-handling mechanism*

Physical Memory Map

The ISA bus physical memory layout is outlined in Figure 9. The memory is broken into three parts: base memory, I/O device memory, and extended memory. RAM is split up on this standard, with a base memory section, holding up to 640 Kbyte of memory, starting at address 0 and ending at the beginning of device memory. Remaining memory is located starting at address 0x100000 (above 1 Mbyte) and extending to as much as 0xFFFFFF (16 Mbytes).

Between the base and extended RAM regions lies device memory, where display adapter cards and LAN cards use special RAM buffers. This region, called the "hole," is a nuisance for UNIX ports, because we would rather see contiguous memory. Although we do have a means of making memory appear contiguous through the use of virtual memory, this does us no good when we must work with physical memory addresses during system bootstrap, hardware DMA devices, and physical memory allocation structures.

```
#include <syscall.h>

        globl _write, _errno

#amtwritten =  write(fildes, address, count);

_write:                                 # caller places arguments on stack
        lea     SYS_write,%eax          # select desired system call
        lcall   $0x7,0                  # call the system
        jb      1f                      # if system returns error, handle
        ret                             # otherwise return

1:      movl    %eax,_errno             # save error in global variable
        movl    $-1,%eax                # indicate error has occured
        ret                             # and return
```

Figure 8: *Code template for the system call stub*

If extended memory is not available, we must temporarily reside in the MS-DOS 640-Kbyte base-memory dungeon. This is truly hell for memory-consumptive UNIX systems. Fortunately, this occurs only when the system is "misconfigured" during the configuration or boot processes, and is not a "normal" situation.

ISA Device Controllers

To support common ISA devices, 386BSD must cope with a separate I/O address bus, shared memory, vectored inter-

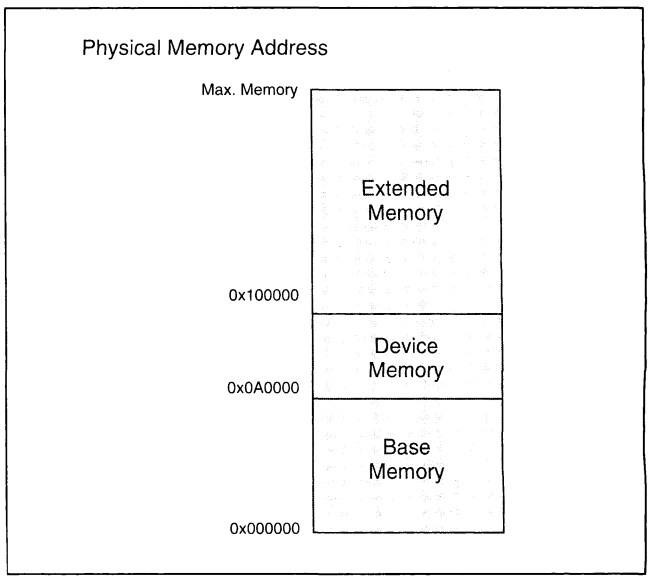


Figure 9: *ISA physical memory map*

rupts, and dedicated DMA controllers. Since most of these evolved from ad hoc standards, device conflicts are common. In order to accurately support ISA, we began with a minimal AT 386 configuration — 386/387, 1-Mbyte RAM, keyboard, monitor, Winchester drive (ST506, ESDI, IDE), and floppy drive — and relied solely on what the BIOS uses to work the hardware. We expect an improvement in performance when these guidelines are eventually relaxed.

ISA Device Auto Configuration

A key advantage of Berkeley UNIX is its ability to configure at boot time devices present on the system. This feature, while difficult to implement on the ISA given numerous conflicts, was considered valuable and was implemented.

In Figure 10(a), we have data structures that encode all the appropriate information to configure a device in 386BSD. Each driver, which may have many devices, is able to locate and configure a device if present. The *isa_device* structure also contains the characteristics of each device to be recognized. If found, hardware resources can then be assigned to each device as configured. A sample table of possible devices to search for within the kernel appears in Figure 10(b).

Interrupt Priority Level Management

In the PC architecture, there is a separate interrupt level per device interrupt. These are more levels than traditional UNIX wants or needs. Instead, UNIX groups different classes of devices into interrupt priority levels that can be disabled and enabled as a group (disks, terminals, network). This is done through *spl( )* function calls, named for a PDP-11/45 instruction which implemented this feature on early UNIX systems. This capability must be provided in 386BSD as well.

Each interrupt vector (interrupt gate) has code that saves the *cpl* (current priority level) variable on the stack, sets the new *cpl* value, and turns on interrupts above this level. On return from the interrupt, all vectors call a common routine that disables interrupts, restores the *cpl*, and returns with interrupts enabled. The *cpl* is altered, as is the priority mask of the dual 8259 ICUs, by the *spl( )* subroutines. The microprocessor or system can now be run at different priority levels on demand.

Bootstrap Operation

One of the last considerations in the development of the 386BSD specification is deciding how we can most easily bootstrap load the BSD kernel from hard or floppy disk. We know that ISA machines have BIOS ROMs that select the device to be booted (typically the floppy first, followed by the hard disk), load the very first block into RAM at location 0x7c00, and finally execute it in real mode. From this point on, we had to create some tight code to run within that 512-byte block to read in our kernel from an executable file in the UNIX file system.

Traditional Berkeley UNIX undergoes a four-step bootstrap process to load in the kernel. First, the initial block bootstrap is brought in from disk by the hardware (in this case, the BIOS). The primary purpose of this assembly language bootstrap is to load in the second 7.5-Kbyte bootstrap located immediately after the initial boot on disk. This larger program, written in C, is much more elaborate in that it can decipher the UNIX file system, extract the UNIX file */boot*, and load it as the next stage in the bootstrap. */boot*, the most complex of the three bootstraps, evaluates the boot event and finally passes configuration parameters to the kernel as it is loading */vmunix*, also located in the file system.

At first we intended to write the initial block bootstrap in

...and There's No Room for Mistakes

A major financial organization needed an automated trading program. Development was projected at six months; it was completed in two – with Action!

An aerospace firm was under critical pressure to create a complex, specialized CAD/CAM package. The smart money said six months. The quiet guys had it wrapped up in less than two weeks – with Action!

A large school district needed a better scheduling system. Designed on the microExplorer, then ported to the Macintosh on Procyon Common Lisp, the new system is powerful, easy to use, and it saved the district six million dollars!

A communications giant needed a feasibility prototype to bid a huge international project. They spent a year trying to develop it. Then, with Action!, it was finished in two weeks.

The Action! development environment used with SPOKE, Procyon Common Lisp and the microExplorer moves computer programming into a vital new dimension. It's helping corporations solve old problems that under conventional methods have defied solution.

Call today to learn more about tough problems that have been solved with the Action! development environment.

The Action! Alliance: a partnership for programming the future.

The Action!
Development System

microExplorer
Lisp machines

ExperTelligence

5638 Hollister Ave, Suite 302
Goleta, California 93117
Phone (805) 967-1797
FAX (805) 964-8448

CIRCLE NO. 465 ON READER SERVICE CARD

SPOKE Object-Oriented
Programming in C and C++

Procyon Common Lisp
with CLOS for Macintosh & IBM

(continued from page 40)

MASM, Microsoft's MS-DOS assembler, and use calls to the BIOS to accomplish the boot process. This proved to be unsatisfactory, as it still left us tied to MS-DOS. So, we decided to use the UNIX protected mode assembler. This allowed us to "cut the cord" with MS-DOS and permitted the system alone to support all code. We also chose to create drivers for the hardware directly, from the initial boot block on up, to break away from the BIOS as well. As a result, 386BSD can now be easily retargeted to new buses that might not rely on either MS-DOS or the BIOS.

(continued on page 46)

(a)

```
/* Per device structure. */
struct isa_device {
        struct  isa_driver *id_driver;      /* per driver configuration info */
        short   id_iobase;          /* Base i/o address register */
        short   id_irq;             /* Interrupt request */
        short   id_drq;             /* DMA request */
        caddr_t id_maddr;           /* Physical shared memory address on bus */
        int     id_msize;           /* Size of shared memory */
        int     (*id_intr)();       /* Interrupt interface routine */
        int     id_unit;            /* Physical unit number within driver */
        int     id_scsiid;          /* SCSI id if SCSI device */
        int     id_alive;           /* Device is present and accounted for */
};

/* Per driver structure. */
struct isa_driver {
        int     (*probe)();         /* Test whether device is present */
        int     (*attach)();        /* Setup driver for a device */
        char    *name;              /* Device name */
};
```

(b)

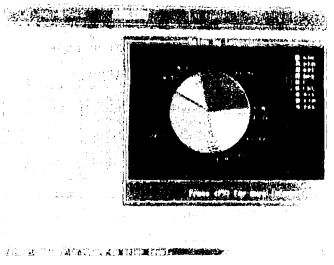
```
/* ISA Bus devices */

#include "machine/isa/device.h"             /* device structure */

/* Software drivers */
#define V(s)        V/**/s
extern struct driver wddriver; extern V(wd0)();
extern struct driver cndriver; extern V(cn0)();
extern struct driver comdriver; extern V(com0)(); extern V(com1)();
extern struct driver fddriver; extern V(fd0)();
extern struct driver nedriver; extern V(ne0)();

/* Possible hardware devices */
#define  C          (caddr_t)
struct isa_device isa_devtab_bio[] = {
/* driver          iobase    irq      drq maddr msiz      intr      unit */

{ &wddriver,       IO_WD0,   IRQ14,   -1,  C 0, 0,         V(wd0),   0},
{ &wddriver,       IO_WD1,   IRQ13,   -1,  C 0, 0,         V(wd1),   1},
{ &fddriver,       IO_FD0,   IRQ6,     2,  C 0, 0,         V(fd0),   0},
{ &fddriver,       IO_FD1,   IRQ6,     2,  C 0, 0,         V(fd1),   1},
0
};

struct isa_device isa_devtab_tty[] = {
/* driver          iobase    irq      drq  maddr         msiz          intr          unit */

{ &vgadriver,      IO_VGA,      0,    -1,  C 0xa0000,    0x10000,      0,            0},
{ &cgadriver,      IO_CGA,      0,    -1,  C 0xa0000,    0x4000,       0,            0},
{ &mdadriver,      IO_MDA,      0,    -1,  C 0xb8000,    0x4000,       0,            0},
{ &kbddriver,      IO_KBD,   IRQ1,    -1,  C 0,          0,            V(kbd0),      0},
{ &cndriver,       IO_KBD,   IRQ1,    -1,  C 0,          0,            V(cn0),       0},

{ &comdriver,      IO_COM0,IRQ4,      -1,  C 0,          0,            V(com0),      0},
{ &comdriver,      IO_COM1,IRQ3,      -1,  C 0,          0,            V(com1),      1},
0
};

struct isa_device isa_devtab_net[] = {
/* driver          iobase    irq      drq  maddr         msiz       intr            unit */

{ &nedriver,       0x320,   IRQ9,     -1,  C 0,       0,         V(ne0),           0},
0
};

struct isa_device isa_devtab_null[] = {
/* driver          iobase    irq      drq  maddr         msiz       intr            unit */

0
};
```

Figure 10: *ISA device controllers: (a) data structures for configuring devices (b) sample table of possible devices*

We slash interface development time.
(and we can prove it!)

C-PROGRAMMERS: See for yourself how Vermont Views™ can help you create user interfaces the *easy* way.

If you want to start saving a *tremendous* amount of time and effort, call for your free Vermont Views demo kit and put us to the test. Vermont Views is a powerful, menu - driven screen designer that comes with a C library of over 550 functions. Which means you can create user interfaces in just a fraction of the time it takes to write the code yourself!



Why try to reinvent the wheel when Vermont Views lets you interactively create pull-down menus, window-based data-entry forms (with tickertape and memo fields), scrollable form regions, choice lists, context sensitive help, and a host of other interface objects.

Vermont Views combines the convenience of a fourth generation language with the power, flexibility, and blinding execution speed of native C code.

Turn your prototype into the application.

Let's face it. With most systems, you have to throw away your prototype when coding begins. Which means you waste precious time and effort. With Vermont Views, things are a lot different. In fact, the prototype actually *becomes the application.* So menus and data-entry forms are usable in the final application without change. Names of functions for retrieving, processing, and storing data can all be specified as the prototype is created. And that's just for starters.

Here's a truly universal solution.

When you create an interface with Vermont Views, you can port it among PC-DOS, OS/2, UNIX, XENIX, and VMS.

Vermont Views can be used with any database that has a C-language interface (most do), and will create interfaces for any roman-based language. Our form-locking version lets you develop quickly and safely on networks and multi-user operating systems, too.

If you need DOS graphics in your applications, we also have the answer. Vermont Views™ GraphEx allows all Vermont Views' windows, menus, and forms to work in CGA, EGA, VGA, and Hercules graphics modes. So you can use your favorite graphics package to create charts, graphs, and other images to enhance text displays.

Vermont Creative Software

Pinnacle Meadows,
Richford, VT 05476
Phone: (802) 848-7731
FAX: (802) 848-3502

WE GUARANTEE YOUR SATISFACTION. FOREVER.

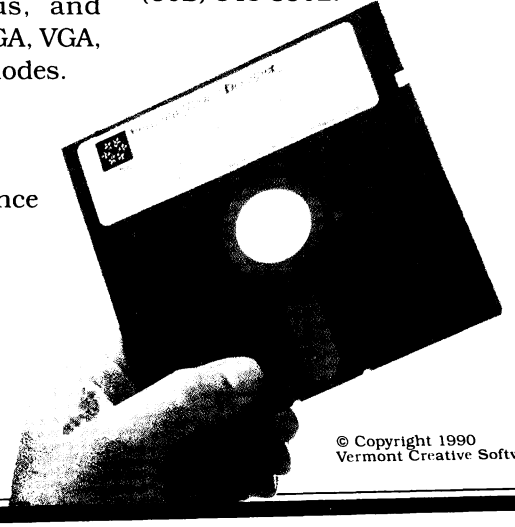
We're so sure you'll love Vermont Views that we make this iron-clad, money-back guarantee. If you're ever dissatisfied with Vermont Views, for any reason, return it for a prompt, no-questions-asked refund. (All you have to do is certify that you haven't incorporated our code into any application.)

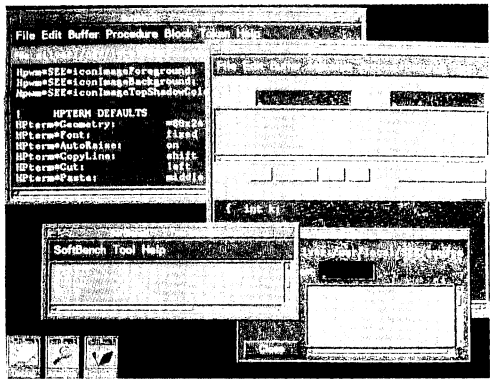
Call for your FREE demo kit!

800-848-1248

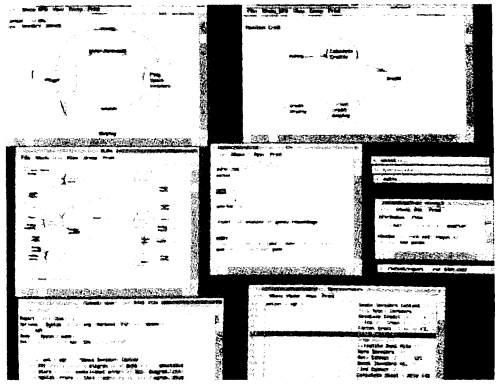
(Please mention "Offer 087")

Don't take *our* word for it. Put Vermont Views to the test by calling for your personal, free demonstration kit. Or fax us at (802) 848-3502.

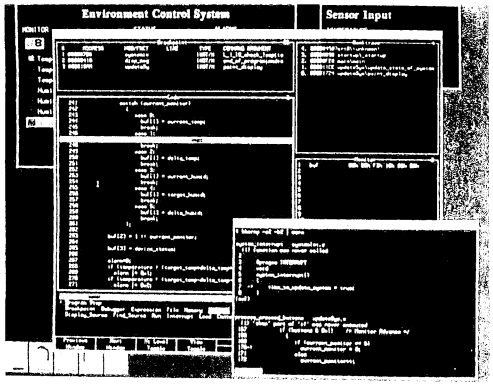




HP C++/SoftBench: A software development environment with an integrated set of program development and integration platform tools.

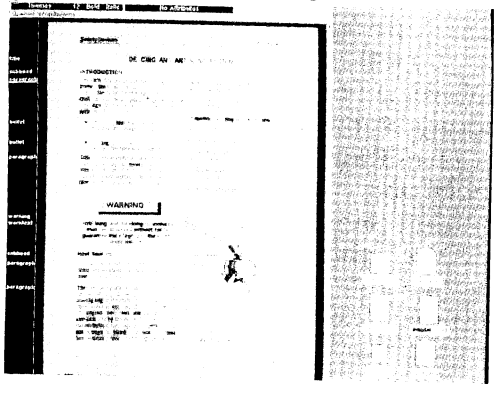


Cadre Teamwork: A family of tools that implement system analysis and software design methodologies.

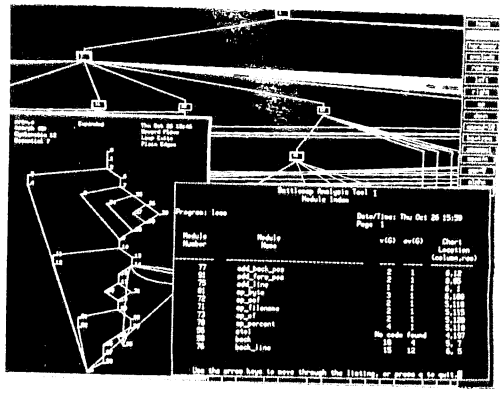


HP AxDB Debugger: Displays microprocessor code, stack backtrace, and variables. Test coverage window shows statements not executed during test.

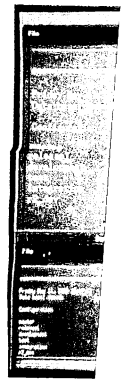
Best CA

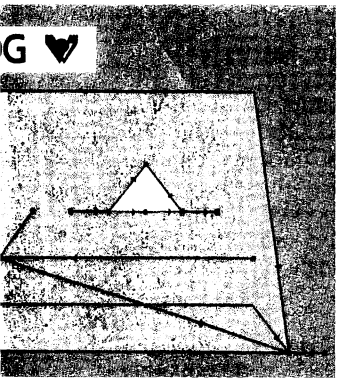


Interleaf Technical Publishing Software: A documentation software and management system that features integrated text and graphics.

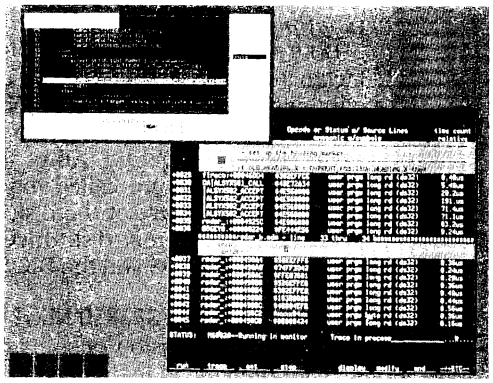


McCabe Test Tools: An automated software testing and reverse engineering application.

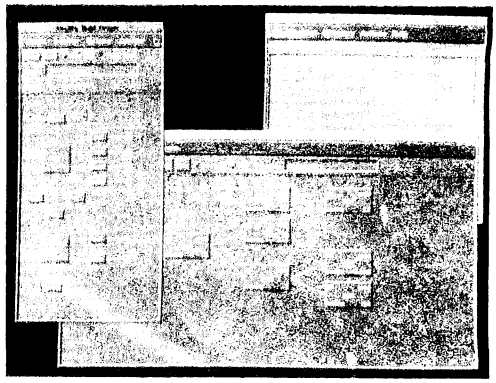




Verilog LOGISCOPE: Automated testing of source code analysis for reverse engineering.

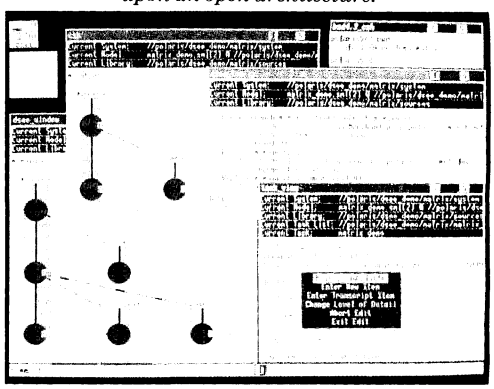


Ada: A development environment that allows real-time symbolic debugging, available from multiple vendors.

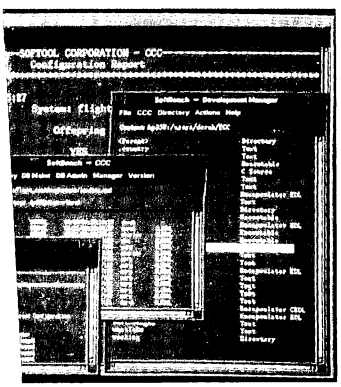


CaseWare® AMPLIFY® CONTROL: A graphic development environment and configuration management system based upon an open architecture.

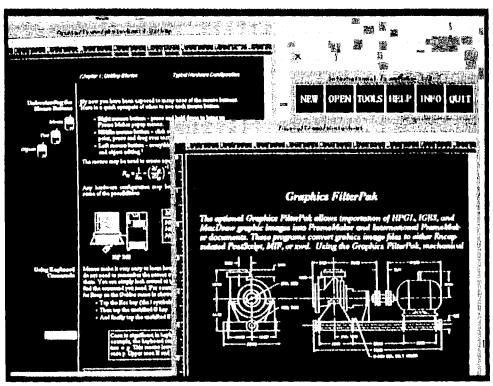
SE scenario.



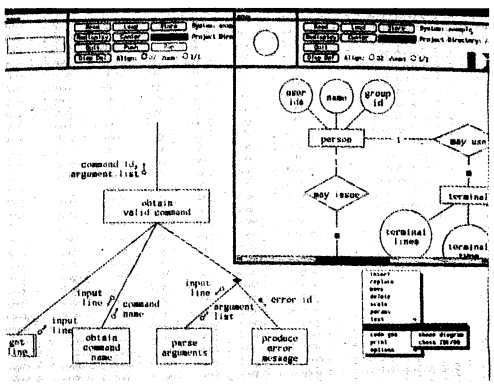
Apollo DSEE: Offers unequaled software development support for complex, team-oriented projects.



Softool Corporation CCC: A complete, automated solution for change control and configuration management.



Frame FrameMaker®: Easy-to-use text, graphics, and layout tools for documentation.



IDE Software Through Pictures®: Integrated tools for improving software quality that emphasize systems analysis and software design.

Your software release dates are continually postponed. Defects are discovered late in your development process. And your team can't find contemporary solutions to existing problems.

Hewlett-Packard CASEdge gives you the competitive edge.

With the HP CASEdge program and our software suppliers, we offer a broad selection of development tools for the software engineering industry. And a strong commitment to helping your team find, evaluate, and implement the right software engineering tools and best practices for the task at hand.

HP CASEdge solutions speed the automation of specification, design, implementation, debugging, and maintenance. And decrease development costs while getting your products to market faster.

For your next design project, choose the vendor with the best CASE scenario. Call HP today at 1-800-752-0900, Ext. 1721.

HEWLETT PACKARD

CIRCLE NO. 513 ON READER SERVICE CARD

(continued from page 42)

Both the second and third bootstraps are actually separate incarnations of the same source code (drivers and all). The only difference is that the second bootstrap is a functional subset of the third bootstrap, so that it could fit within the small confines required. All of the bootstraps reference a special data structure called the *disklabel* that knows the layout and geometry of the disk drive booted. In this way thousands of different disk drives can be supported independent of MS-DOS and the BIOS information.

Summary: Where is 386BSD Now?

Perhaps the discussion of some of these issues might have seemed difficult or incomplete, but we found each item to be of tremendous importance in understanding the practice of a port to the 386 architecture. Unlike Berkeley UNIX ports to other systems, we found that we had to bend over backwards dealing with segments, memory issues, device issues, and a plethora of unique microprocessor features. Now, one may ask, was it all worth it?

Well, BSD is now available on the 386 platform. Even though it is only a preliminary release, we already support the following:

- Many different PC platforms, including the Compaq 386/20, Compaq Systempro 386, any 386 with the Chips and Technologies chipset, any 486 with the OPTI chipset, Toshiba 3100SX, and more.
- ESDI, IDE, and ST-506 drives
- 3-1/2 inch and 5-1/4 inch floppy drives
- Novell NE2000 and Western Digital Ethernet controller boards
- EGA, VGA, CGA, and MDA monitors
- 287/387 floating point, including the Cyrix EMC
- A single-floppy standalone UNIX system, containing support for modems, Ethernet, SLIP, and Kermit to facilitate downloading of 386BSD to any PC over the INTERNET network.

Those of you who can meet University of California requirements should obtain a copy of 386BSD from the University of California, so that you can follow along yourself as we work through the basics of this port from every angle.

In addition, we would like to thank some of the people who have helped make 386BSD a reality, including Mike Karels, Keith Bostic, and Kirk McKusick of CSRG, Dixon Dick and all the support engineers at Compaq, Fred Dunlap and Bob McGhee of Cyrix, Don Ahn (UCB), Tim L. Tucker (Evans and Sutherland), and Clem Cole (Cole Computer Consulting).

Suggested Readings

1. Leffler, Samuel J., Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Reading, Mass.: Addison-Wesley, 1989.

2. Crawford, John H., and Patrick P. Gelsinger. *Programming the 80386.* Alameda, Calif.: Sybex, 1987.

3. *IBM Technical Reference: Personal Computer AT.* Boca Rotan, Fla.: IBM, 1984.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**