386BSD: A Look Under The Hood.

Andrew McRae

Megadata Pty Ltd.
2/37 Waterloo Rd
North Ryde
andrew@mega.com.au

386BSD is a freely available port of the BSD Networking Release 2 software to a 386 PC architecture. It is a fully functional kernel with associated user programs and tools, albeit not yet a commercially stable release.

Initially the history and background of this system will be explored, then some machine specific aspects of 386BSD will be discussed, such as system startup, I/O configuration, and the virtual memory subsystem. Some discussion of various performance and porting issues will follow, and some conclusions drawn concerning the strengths and weaknesses of this particular UNIX port, and how well it fits onto the 386 and the PC hardware architecture.

Disclaimer: *This paper does not represent the views of AUUG or Megadata, it contains solely my own (sometimes tongue-in-cheek) opinions.*

Introduction.

1992 saw the release of a freely available BSD port to the ubiquitous 386 PC architecture, based on the BSD Networking Release 2 and ported in the most part by William Jollitz. Whilst not considered a commercially supported UNIX release, it is a fully functional BSD system, with networking and multi-user support included. It has provided access for many people to a working kernel with source, which has in the past been unobtainable except to sites that have licenced the official AT&T sources.

In the course of researching some hardware profiling techniques, I used 386BSD as a case study to examine the performance of a typical kernel. As a result of this study, I delved somewhat into the internal structure of the kernel. This paper describes some of these internals, and also attempts to highlight interesting areas where improvement may be obtained.

History and Background.

A number of operating systems have been released for free use, among them Linux, Minix etc. Most of these are aimed at conforming to POSIX functionality, so that they follow a standard Application Programmer Interface and User Interface. They generally differ from UNIX in that they tend to be written from the ground up having no access to AT&T source code. On the other hand, the BSD networking releases have been developed as part of kernels incorporating portions of the official AT&T source code of UNIX, and there has been a degree of sharing of code between the two organisations, such as the networking software, the BSD Fast File System etc (some people feel that there has been *too* much sharing of code).

Historically, there have been several releases of software from the Computer Science Research Group (CSRG) at the University of California at Berkeley. These distributions were called Berkeley Software Distribution (BSD) releases, and provided major building blocks for vendor kernels in the area of networking and file systems, and were a key element in the growth and prevelance of UNIX in the workstation and distributed computing environment. A number of major vendors based their kernels on BSD 4.2, which itself was based originally on the 32V UNIX† VAX version from AT&T; thus all complete releases of

† UNIX is a trademark of Bell Laboratories.

working BSD kernels required at least a 32V UNIX source licence. More recent versions since 4.2 BSD have reworked the networking areas and added new ports to different architectures (4.3 and 4.3Tahoe).

A side effect of this continuing development by the CSRG has been the replacement of the AT&T portions with freely available unencumbered software, so that more and more sections of the kernel may be released as part of BSD software releases. The first of these releases (of a limited distribution) was 4.3Reno. The goal was to eventually release unencumbered complete kernel source and utilities for a range of architectures. The VAX (and CCI) architecture ports were to be deprecated, and newer ports to Motorola 68K and Intel 386 architectures were to be integrated and supported. Whilst the CSRG were not directly involved with some of these ports (such as the Sparc port, done by Chris Torek at Lawrence Berkeley Laboratories), it was planned to release ports for commonly available hardware. The CSRG supported the 68K port (originally from the University of Utah) on the HP 300 architecture; William Jolitz provided a port to the Intel 386, based on the IBM PC architecture.

Aside from porting, new functionality was added in the form of a NFS implementation (from the University of Guelph), implementation of the lower layer OSI protocols (from the University of Wisconsin), and other ISO upper layers and applications based on the ISODE distributions.

The next planned major release was to be BSD 4.4, but in the meantime an interim release was generated that contained the work to date, termed the BSD Networking Release/2 (commonly referred to as NET/2). NET/2 contained significant functionality, but with some key modules missing. It was becoming clear that the role of CSRG was changing, and it was seen that the release of 4.4 BSD would also spell the demise of that group.

The release of NET/2 in 1991 sparked a number of events; since it was a system that was *almost* there, people saw an opportunity to use NET/2 as the basis for a real operating system that was free of licensing restrictions. A company known as Berkeley Software Design Inc. (BSDI) was formed with the goal of taking the NET/2 release and creating a commercially viable and supported system that could be sold with source code. The initial product used the Intel 386 port based on the ubiquitous IBM PC architecture. This product is known as 386/BSD. Latterly this company has obtained the SPARC port code, so in the future it is likely that other popular architectures will be supported.

The Legal Situation.

The exciting prospect of finally achieving truly open and available systems (the ‘Prague Spring’) came to a jarring halt when tanks rolled in driven by men in dark suits carrying loaded legal injunctions, declared war on BSDI for infringement of trademark, then for copyright infringements; the embattled BSDI unexpectedly gained a co-combatant when Unix Software Laboratories (USL) extended the suit to the Regents of the University of California at Berkeley. The members of the CSRG spent considerable time writing legal briefs instead of operating system software, delaying the expected release date of 4.4BSD, though at the time it was not clear whether there was going to *be* a release of 4.4BSD.

A major premise of the law suit was that the unencumbered portions of the software was written by people who were exposed to copyrighted code, and therefore were (in the minds of the lawyers) ‘mentally contaminated’ by the code, and any subsequent code they produced should be influenced by copyright. Many people have argued that the only way programmers could be mentally contaminated is by writing BASIC programs, and that no-one would want to copy the sort of code that is contained within the USL distribution anyway, but it is unclear what the courts will make of this.

There has been much made of the legal and political ramifications caused by the various suits, and it is likely that whilst battles may be won or lost, the war may go on for quite a while.

Recently BSDI won an injunction preventing distribution of a first release of their product, so BSDI is now allowed to sell non-beta versions of BSD/386. Now that the genie is out of the bottle, it is difficult to see how he can be forced back in.

The Role of 386BSD.

386BSD is another example of a system based on the NET/2 release. William Jolitz (who did the initial 386 port) filled in the missing pieces to create a 386 based BSD system, and initially released it to the world as a freely distributable system in early 1992 as version 0.0. This allowed access to a working ‘real’ kernel for the great unwashed masses (such as myself), so that development, testing and learning can take place (generally known as 'kernel hacking'). William Jolitz documented a large portion of this work in a series of articles in *Dr Dobbs Journal*, and is planning a book to be released later this year called **386BSD From The Inside Out**.

The primary and stated goal of 386BSD was to provide a *research* tool for people to experiment with new operating system ideas, or to examine the internals of a functional system to see how it runs.

After an early phase of instability due to the immature state of the system, a much more reliable release was available in July 1992 (version 0.1). This contained many contributions such as a PC filesystem, CDROM support, SCSI support etc.

There is an emerging view of 386BSD in that it is a social experiment to see if everybody 'on the net out there' can actually support, contribute and improve what is in effect freely available common property. Already there is a growing cadre of core users who have contributed in major ways such as supporting the X Window system, coordinating contributions and fixes, and improving the basic functionality by implementing new features such as shared libraries etc. Time will tell just how effective this will be.

My Interest in 386BSD.

Having developed a hardware profiling technique, I wished to use this to closely examine the performance aspects of a real kernel, and with the advent of 386BSD this became possible. In doing so, I discovered some interesting things about the internals of 386BSD and the PC architecture, and this paper's goal is to shine light into some of the dark corners. I will examine some of these areas, and then discuss the profiling results.

System Startup.

Booting 386BSD is a two stage process. The BIOS code on a PC attempts to load a boot loader by reading the first sector on a floppy or hard drive attached. This code initialises enough of the 386 memory descriptor tables to allow the loading of the secondary 386BSD boot loader, which is contained on the next 15 blocks (7.5 Kbytes).

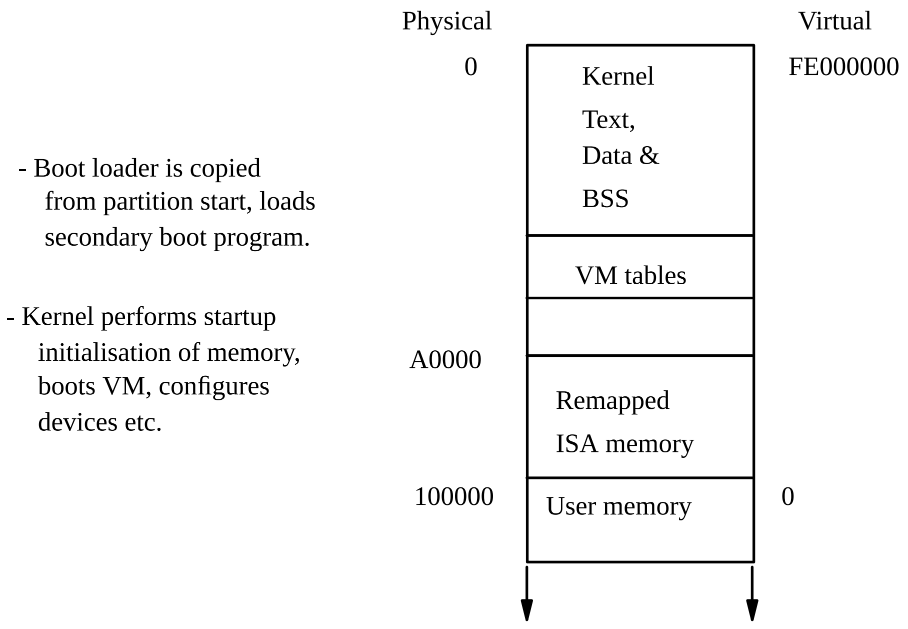
Physical
Virtual
0
FE000000
Kernel
Text,
Data &
BSS
- Boot loader is copied from partition start, loads secondary boot program.
VM tables
- Kernel performs startup initialisation of memory, boots VM, configures devices etc.
A0000
Remapped
ISA memory
100000
User memory
0

Figure 1 - Memory Arrangement

This secondary boot loader understands enough of the file system format to search for and read in the main kernel file, which is then stored in the lower 640 Kb section of the PC memory. After control is transferred to the 386bsd kernel, the physical memory addressing is remapped to new virtual locations as shown in figure 1.

The default name searched for is *386bsd,* and the file is linked to run at an address of 0xFE000000. Currently the kernel is limited in size so that it fits into this lower portion of memory; future versions will remove this restriction.

In effect, the kernel is remapped to absolute location FE000000; the last location of the kernel is rounded to a page boundary, and a fixed number of pages are allocated for the kernel stack, a proto u-dot area and other virtual memory requirements. The ISA memory address space is then remapped to follow this kernel address space; the virtual address that this memory is mapped at may vary depending on the size of the kernel.

Once the VM and exception handling is bootstrapped, the system startup looks very much like other BSD based kernels, where configured devices are probed to determine the I/O configuration, and system initialisation takes place such as handcrafting the *init* process, and setting up the system page maps etc.

Virtual Memory on the 386.

The 386 has a modern paging virtual memory architecture that uses a two level page table scheme to map fixed size memory pages of 4 Kbytes each. Figure 2 shows the basic structure of this memory arrangement. In fact the 386 also allows extra segmentation descriptors to reference beyond the 32 bit address space, but quite rationally the 386BSD VM subsystem fixes the segment descriptors so that a flat 32 bit addressing mode is used.

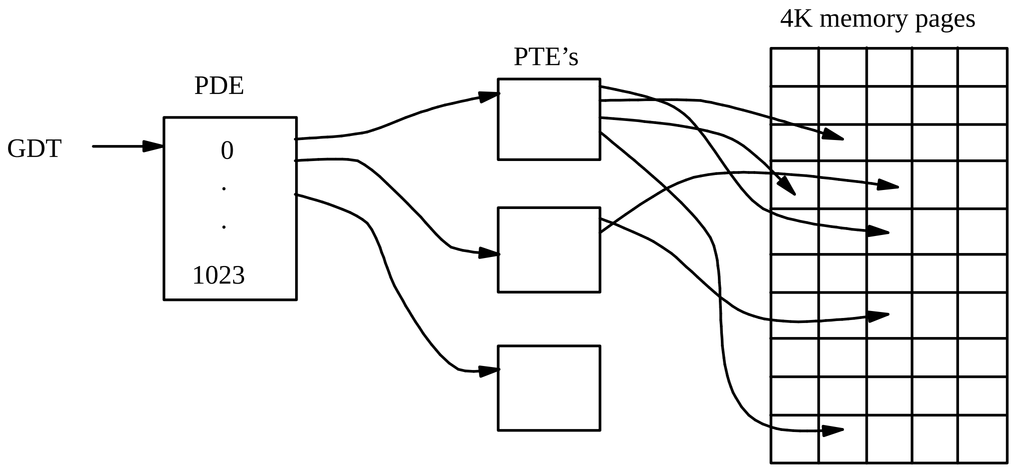

Figure 2 - 386 Virtual Memory

Allocated at the end of the kernel BSS space, the Page Directory Entry (PDE) table is a 4Kb page that has 1024 entries pointing to separate 4Kb Page Table Entry (PTE) pages, each of which holds up to 1024 Page Table Entries, each referencing one 4Kb page of memory. Each Page Table page itself is not required to be present in memory i.e it may be paged in as required, or not allocated to leave 'holes' in the VM address space. Thus the 32 bit flat address is divided into 10 bits of index into the Page Directory entries, the next 10 bits indexing the PTE within the page pointed to by the PDE; finally the remaining 12 bits reference the byte offset within the memory page referenced by the PTE.

To reduce the amount of information that must be obtained from memory to actually process a memory access through the VM subsystem, the 386 has a 32 entry Translation Lookaside Buffer (TLB) that caches PTE references.

Earlier BSD systems used a VM architecture based on the VAX virtual memory system; this was orientated around smaller and more expensive memory, and fast disc I/O. More modern systems are designed around larger memories and slower peripherals, so the Mach virtual memory subsystem (originating from Carnegie Mellon University) was grafted into the BSD kernel. This VM architecture centred around architecture independant page maps, where a system dependant portion would be implemented to map the the desired actions onto the particular hardware VM architecture in use. This allowed sophisticated features to be added such as copy-on-write, sharing of memory and mapping of files into memory, but without forsaking portability. The system dependant code was termed the *pmap* module for that architecture.

So to operate the Mach VM architecture a 386 specific *pmap* modules exists, which acts as the intermediatary (when VM changes need to take place) between the standard VM code and the the physical manipulation of the 386 page tables. This module is a critical section of the 386BSD implementation, as it is a major part of the system dependant portion of the operating system. Since under a multi-tasking, multi-user paging operating system like 386BSD much of the processing of the system is taken up with the manipulation of the virtual address space, it is vital that this interface module be as fast and efficient as possible.

The kernel is linked to execute at the virtual address of (hex) FE000000, and the physical memory is mapped accordingly. User programs are linked to begin execution at virtual location 0, and memory allocated on a page by page basis as is required for the program. The linking of the kernel at high memory allows the kernel mode of each process to coexist within the same virtual address space as the user mode of the process, allowing direct access to the user data space by the kernel code. Transfer of data across the user/kernel boundary can then be simply a data move, allowing much higher transfer rates across the boundary, which in earlier systems was a major system bottleneck.

I/O Subsystem.

Currently only the ISA peripheral bus is supported; drivers exist for most commonly available boards, such as SCSI disc controllers, IDE controllers, a range of network cards, and serial cards. It is expected that a future release will incorporate support for EISA bus peripherals.

When the system is first initialised, the configured devices are probed to check their existence, and to allocate their interrupt vectors etc. Some devices are memory mapped in the so-called I/O memory physical address space from (hex) A0000 to FFFFF. When the kernel is initialized, this I/O memory is mapped to the end of the kernel space in virtual memory addressing, and the I/O memory address allocated at configuration time is adjusted accordingly.

One of the reasons why the PC architecture is so popular is because the hardware is plug compatible, and motherboard support for DMA, interrupts etc. has not changed since their introduction. This is a double edged sword; since the hardware bus architecture is the same as 10 years ago, drivers can be assured of compatibility, but that architecture has not scaled well with faster memories and processors. Compatibility is assured, but with devices that were once discrete ICs; whereas now nearly all functionality can be contained within a small number of fast VLSI chips, but these must still obey the slow timing rules of the original design.

One example of this is the interrupt structure. Originally the 8086 processor only allowed one interrupt line, and there was a single bit mask to disable this interrupt. Separate Interrupt Controller Units (ICUs) were used to provide multiple sources and vectoring, and the processor used I/O commands to program the ICUs with the desired interrupt masks. This has been carried through into the very latest processor designs, and the same interface is required to program the ICUs to allow/disallow different classes of interrupts. This adds overhead and complexity to the handling of driver interrupts.

Another example is the use of DMA over the ISA bus. Since the address bits used on the bus are limited, systems that have large amounts of memory cannot DMA into high memory; this requires the use of a *bounce buffer* in low memory, and CPU intervention to copy the results of the I/O into the desired final location.

Results of Profiling.

After profiling a number of the key areas of the kernel, some impressions emerged concerning the kernel performance. These fall into three main categories; CPU performance, I/O performance and virtual memory management. The platform was a 40 MHz 386 with 64 Kb cache and 8 Megabytes of main memory.

Firstly, I was pleasantly surprised to note the oft maligned Intel architecture did indeed run fast, especially at a clock speed at 40 Megahertz and employing 64 Kb of external cache. Moving data through the kernel to user space was faster than expected, and it was clear that function call and return was also speedy.

Undoubtedly memory speed and cache effects have a major impact on performance, as data throughput dropped markedly whenever memory was accessed on the ISA bus as opposed to main memory. More on this later. Profiling the interrupt code showed that the regular clock tick interrupt took on average 94 microseconds to execute; unfortunately the hardware architecture does not provide for Asynchronous System Traps (commonly known as software interrupts), so the interrupt code has to work extra hard to emulate this facility. The interrupt code overhead to do this is around 24 microseconds per interrupt; it is hard to judge whether this has a significant impact on system performance.

Due to the interrupt architecture of the bus and the processor, it was evident that more time was spent ensuring correct synchronisation and interrupt lockouts than would normally be required on a multi-priority interrupt level processor such as 680x0; on the average it took 11 microseconds per *splnet* call, which may not seem a long time, but the *spl** routines get called a great deal, and it all adds up to a significant amount. In one test, 9% of the total CPU time was spent in *splnet, splx, splhigh* and *spl0*. Unfortunately it is hard to see how this could be improved, given the nature of the interrupt architecture.

When some tests were performed where input/output activity was heavy, it was clear that a major bottleneck in system performance is the use of the ISA bus. This was especially noticeable on the Ethernet adaptor, which is a 8 bit wide controller. To transfer similar amounts of data, the ISA bus is up to 20 times slower than main memory transfers.

It would be instructive to profile different controller cards to determine where each performed best; when support for EISA cards is available it would be interesting to see what performance gain would be obtained using the higher bandwidth bus.

Whilst the CPU performs reasonably well, overall performance is crippled by the poor I/O bandwidth, and the interrupt architecture of the 386 and the ISA bus also contributes to reduced performance.

The virtual memory management subsystem of 386BSD was derived from the Mach memory management code. Following code path traces of various virtual memory functions indicates that the VM subsystem is definately non-optimal. Some functions seem to run surprisingly fast; the routine that handles page faults and enables new pages to be accessed (*vm_fault*) takes about 400 microseconds, which seems reasonably low overhead. On the other hand, an excessive number of page faults seem to occur at times. Where the real performance problems lie is in creating new VM contexts for new processes, as explained in the next section.

Fork/exec Profiling.

A common operation of UNIX is to *fork* a process and create a child copy of the process, which then *execs* a new process image. For UNIX to perform well, these two operations must be reasonably fast, since some UNIX operations rely on a low cost of process creation; shell scripts for example rely on fast execution of processes to achieve a reasonable performance level. Due to the portable nature of shell scripts, it is becoming more and more common to employ shell scripts instead of compiled binary programs.

The current situation looks fairly abysmal; it takes some 24 milliseconds to perform a *vfork* operation, and it takes about 28 milliseconds to perform an *execve* system call. This adds to about 52 milliseconds to perform a combined fork/exec operation. Note that these times do not include any disk activity, as the process image was already cached. Where is this time being used? In figure 3 a summary of the highest cost subroutines is shown.

Most of the CPU time occurs within a small number of routines; it is clear that the *pmap* module is a bottleneck when manipulation of the virtual memory is required (the *bcopyb* call relates to scrolling of the console screen, so it should be ignored for the purpose of the exercise). Over 50% of the time is being spent in the virtual memory routines shown above. Examination of the code path trace shows that *pmap_pte* is called 1053 times when a *fork* is executed, and a similar amount when an *exec* is done. Further analysis of the code path shows the exact progress of the fork operation, and each subsection can be examined in detail to see the amount of time it is taking, and whether significant optimisation can take place. There is a major amount of cross-calling between the *pmap* module, and the rest of the virtual memory subsystem, so it is envisaged that a major performance benefit would occur if some of that glue could be trimmed back and some sculpting of the interface performed.

```
Elapsed  Net   # calls  (max/avg/min)  % real  % net     name
77603  58913      67    (14061/879/2)   5.02% 28.22%   pmap_remove
22283  22148    5549         (66/3/2)   1.89% 10.61%   pmap_pte
12938  12938    1215        (13/10/9)   1.10%  6.20%   splnet
10912  10874       3 (3634/3624/3613)   0.93%  5.21%   bcopyb
33435  10134     453       (40/22/21)   0.86%  4.85%   spl0
15963   7876       8     (3862/984/3)   0.67%  3.77%   pmap_protect
 5657   5657      77       (244/73/3)   0.48%  2.71%   bcopy
47723   4889     115       (64/42/27)   0.42%  2.34%   vm_fault
 4759   4759    1349          (5/3/3)   0.41%  2.28%   splx
 7836   4361     236       (29/18/13)   0.37%  2.09%   vm_page_lookup
 7320   3489     119       (39/29/12)   0.30%  1.67%   pmap_enter
 3457   3457      38       (132/90/2)   0.29%  1.66%   bzero
```

Figure 3 - Fork/Exec Summary

Network Performance.

Profiling was performed on the TCP/IP and socket code by running a program that listened on a socket and when another host connected, read and discard the data. A Sun Sparcstation 2 was used as the host to send the data, as I was sure it could fill the available network bandwidth to the PC over an ethernet.

This was the only test that caused the PC to be totally CPU bound, so that essentially the CPU was busy 100% of the time. It was obvious that the PC could not process the data from the network at anywhere near Ethernet speed. Examining the code path trace and function summary showed that 33.6% of the time was spent in *bcopy*, and that 30.8% of the time was spent in *in_cksum*. Again, *splnet*, *splx* and *spl0* contributed around 9% of the time.

Delving further into the code path trace, it was clear that a major bottleneck occurs because the Ethernet driver for the card must copy that data from the onboard controller memory across the bus; each TCP data packet that was received (i.e a full Ethernet packet) took about 1045 microseconds to process at the driver level. This alone is only about 20% more data throughput than Ethernet itself, so it is unlikely that Ethernet data rates through to the network applications can be achieved using this 8 bit controller card, unless the rest of the software has been tuned for minimum overhead.

The other major CPU user was the checksum routine itself, which was almost a big an overhead as the driver packet copy. This was surprising at first, as the packet was now in main memory, and the checksumming should be close to memory-to-memory copying speeds. To checksum a 1 Kbyte packet was taking 843 microseconds. It was discovered that the *in_cksum* routine has not been optimally coded (e.g like other architectures where it is done in assembler), and recoding this routine should provide a reduction in packet processing from 2000 microseconds to perhaps 1200 microseconds; this would provide a major improvement in network performance, and the limiting factor would become the memory bandwidth available to the network controller across the ISA bus.

Another conclusion that can drawn is that a much faster I/O architecture is required before serious data throughput can be expected, but I think we all knew that.

Filesystems.

Separate profiling studies have been performed on the BSD Fast File System (FFS) code and the Network File System code. Due to the network performance problems discussed in the previous section, any performance issues in the actual NFS implementation are totally swamped by the I/O bandwidth limitations. An interesting situation arises due to the fact that UDP checksums are usually turned off with NFS; since the checksum routine contributed a large proportion to the CPU overhead, NFS actually provides less overhead and better throughput than an FTP style connection!

The disc controller used in the target PC was an IDE controller on a Seagate ST3144 disc. The FFS profiling showed how disc seek times impact the I/O throughput. Each read of the disc varied from 18 milliseconds up to 26 milliseconds. Each write interrupt took about 200 microseconds in total, with about 149 microseconds of that being actual transfer time of the data to the controller. Interrupts seemed to be close together most of the time (< 100 microseconds), so the disc driver may well be improved by waiting a short time after transferring the data to see if the controller is ready to accept another block straight away.

Overall, the CPU was only busy for 28% of the time when doing a large number of writes, so the disc seek times are still the major influence in determining disc throughput. It was interesting to see that out of that 28%, at least 6% was spent in the *spl** routines. It would be interesting to use a different type of controller (maybe one with DMA) and see what difference it makes.

Where Do You Get It?

There are a number of sites which contain the official distribution, as a well as maintaining mirrors of the unofficial contributions, drivers etc. In Australia, **kirk.bu.oz.au** is an official mirror of the semi-official central repository of 386BSD software **agate.berkeley.edu**, run by Chris Demetriou (*cgd@cs.berkeley.edu*). Jordan Hubbard (*jkh@whisker.lotus.ie*) is the current co-ordinator of the 386BSD patchkit; patchkit is an automated method of applying bug fixes and patches, as well as the accumulated patches. It is highly recommended that these patches are applied.

The Usenix news groups **comp.os.386bsd.{apps, questions, development, bugs, misc}** cover the usual range of discussions, arguments, complaints and misunderstandings. Compared to the usual level of support that users get for operating systems that they pay for, the Net actually seems to provide fast and accurate results (some of the time, anyway).

It is expected the the offical 0.2 version will be released sometime Real Soon Now; it is likely that this release will integrate the patches and bugs already reported, as well as providing new features such as EISA support and shared libraries.

Conclusions.

The major conclusion about the performance of 386BSD is that there are a small number of areas that need addressing, that when fixed should improve the performance considerably. The hardest area to address is the virtual memory subsystem. The easiest area would the IP checksumming. The grossest area of mismatch between the hardware architecture and UNIX is the interrupt priority control and lack of software interrupts.

It was also clear that the hardware I/O performance is a major factor, and that the platform the profiling was performed on is crippled in I/O bandwidth.

How well does UNIX fit on a PC architecture? There are a number of areas where the fit is a bit rough; this is especially true in the I/O area, where the lack of a decent interrupt structure adds considerable overhead, and the poor performance of the peripheral bus limits the bandwidth of networking and mass storage data transfer. This begs the question, what is the difference between a (real) workstation and a PC? The answer seems to be I/O bandwidth and overhead, since the memory capacity and CPU performance of PCs seem to be reaching similar levels. The biggest strength of the PC architecture, that of hardware and binary compatibility, has also seen it chained to an architecture that is over a decade old, and is showing its age.

The future bodes well, however, for 386BSD. It is likely to provide a reasonable platform for teaching, research and experimentation for quite a while; at least until perhaps Plan 9 is freely available.