An Appraisal of the Instrumentation in Berkeley UNIX 4.2BSD

Michael David Kupfer

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Berkeley UNIX† 4.2BSD comprises four subsystems: networking, the file system, virtual memory, and process management. We present explanations of how those subsystems work and problems with their instrumentation in 4.2BSD. We discuss inaccuracies in the reported performance indices, cases where uninteresting indices are recorded, and cases where interesting indices are not recorded. We restrict our attention mainly to tools that sites can use without special hardware and without their changing the UNIX kernel. We also suggest tools for debugging and monitoring the networking and file system subsystems. Last, we comment on general implementation issues at both the kernel and user levels of the instrumentation code.

"The trouble with paper designs is you don't get bloody enough."

Butler Lampson

†UNIX is a Trademark of AT&T Bell Laboratories.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N0039-84-C-0089. Additional support was provided by the State of California MICRO program, grant number 532422-19900. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency, the US Government, or the State of California.

1. Introduction

We shall first explain the goals and recognized shortcomings of this paper. Then, as the instrumentation of "tunable resources" is very important in this project, we shall discuss how to tune kernel variables in 4.2BSD. The last part of the Introduction previews the rest of the paper.

1.1 Goals

This project began with the notion that bugs had crept into the user-level and kernel routines that instrument a Berkeley UNIX 4.2BSD system. Rather than embark on a massive debugging binge, it seemed more profitable to consider UNIX instrumentation as a whole: get rid of fluff in the current code; devise schemes to better monitor the system, especially in those parts that had changed the most since Version 7 UNIX; and improve the engineering of the tools. For two reasons we did not want to require of our users access to UNIX source code. The first reason was to make the project useful both for sites with source access and for sites with such binary-only licenses as Digital Equipment Corporation's ULTRIX-32™ license [Ultrix]. The second reason was simply to streamline the project by ignoring tools like **gprof** and **kgmon** [UserManual]. Many of the tools that we consider in this paper deal with the use of system resources like memory, disks, or network connections. Most of the remaining tools help system managers tune system parameters like the maximum paging rate or the size of kernel tables.

In addition to studying the existing instrumentation, we shall also suggest tools to help debug or otherwise monitor two major changes introduced in 4.2BSD: Interprocess Communication (IPC) [Leffler *et al* 83a] [Sechrest] and the Fast File System [McKusick *et al* 84]. The IPC tools are needed to support a vast library of new and complicated code, whose effect on the system is not fully understood. Also, writers of distributed applications need better tools for their work. Similarly, better tools are needed to understand the effects of the new file system.

We should also make clear what the limitations of this project are. First of all, we are presenting a paper design, not an experience report. Another limitation is that we desire no hardware support, mostly so that our tools will be universally useful. Third, we are not designing an expert system for configuring UNIXes. That is, the tools will not recommend steps to take to improve system performance. Finally, we are not considering tools like **time** or **gprof**. These programs instrument other programs, rather than the system as a whole.

1.2 Caveats

The title of this report is misleading. We are not really presenting an instrumentation package for 4.2BSD, even though we shall say "4.2BSD" throughout the paper. In reality we are presenting a package for Berkeley UNIX as it existed some time after the 4.2BSD distribution tape was made, but before the system was frozen for 4.3BSD. Thus, certain bugs in the 4.2BSD release will not be considered because the project started after those bugs had already been fixed at Berkeley. More important, the paper is trying to hit a moving target. It is possible that future implementors will find that parts of the paper are inaccurate or no longer relevant because of subsequent changes to Berkeley UNIX.

Another important point is that this paper does not present a complete design. Implementors will need to build some prototypes, either to verify the usefulness of certain tools or kernel modifications, or to ensure that the new code will not substantially disturb the system being measured. Finally, we make no guarantees that we have found all the bugs in existing instrumentation tools, nor do we claim that the list of enhancements is exhaustive.

1.3 Tuning the 4.2BSD Kernel

Because we shall be talking about instrumentation of tunable kernel parameters throughout this paper, we'll explain how to tune those parameters. Tuning gets done either in the system's **config** file

[Leffler], by manually changing the initial values of kernel variables, or by changing the definitions of symbolic constants in the kernel. When you first build a kernel (that is, the first time you type "make"), the file **/sys/conf/param.c** is copied into the directory that you're building the kernel in. **Param.c** initializes certain variables (e.g., **nproc**, **nmbclusters**) based on values in the kernel's configuration file and on some default rules. In fact, many system tables are sized according to the **maxusers** entry in the **config** file. If you don't like the default rules, you can edit **param.c** and then rebuild the kernel. You can change other initial values by editing kernel source code. This technique works both for variables (e.g., **nbuf**) and symbolic constants (e.g., **SYSPTSIZE**). Some variables, such as **nbuf** and **bufpages**, are checked when the kernel starts up. If the variable is zero, then the kernel computes a value for it based on default rules. If the variable is non-zero, then the kernel leaves it alone. Thus a third way to change initializations is to patch the kernel object file using **adb**.

1.4 The Rest of the Paper

For this project, we break down Berkeley UNIX into four subsystems, corresponding to four major system resources: networking and IPC, I/O and the file system, virtual memory, and processes and the CPU. We present IPC first because it is the newest and least instrumented of the subsystems. We next present I/O and the file system. This area is also new, but it seemed better instrumented and understood than the IPC code. Third is virtual memory, which has been around for a relatively long time (the Berkeley paging kernel was released in 1979), but whose instrumentation appeared cryptic. Last come processes and the CPU. This paper also examines the 4.2BSD accounting package. We consider accounting to be system instrumentation because commonly used programs (e.g., the **vi** editor) will impair the entire system's performance if they themselves make heavy use of system resources. Although the accounting package covers all four subsystems, we have arbitrarily placed it in the CPU section.

Each subsystem has its own section of the paper. Each section is itself divided into three parts. The first part gives a brief, general explanation of how that subsystem works. The second part explains what measurement tools already exist for that subsystem. If necessary, it explains in more detail how the subsystem works. The third part critiques the existing tools, explains bugs that we found, and suggests improvements. Particularly detailed explanations and suggestions will be given in smaller type, usually in a separate paragraph.

Section 6 of the paper contains general comments on implementing the ideas presented in the paper. In particular, we shall discuss validation suites, methods for collecting and displaying information, and software engineering. The Appendix lists the routines discussed in the paper, what sections they're discussed in, and what bugs were found in them.

2. IPC and Networking

2.1 How It Works

The 4.2BSD IPC system was designed to serve as a test-bed for research in protocols and distributed computing. This goal led to a 3-layer structure that we shall explain below, based on *sockets*, *protocol families*, and *interfaces* [Leffler *et al* 83b]. Other interesting properties of the IPC code are:

- It does its own memory management (using a data structure called an *mbuf*)
- Routing is done on a system-wide (rather than protocol family-wide) basis
- The routing policy is implemented by a user process rather than in the kernel

We shall explain more about these issues shortly. We shall also comment on the Defense Advanced Research Projects Agency (DARPA) *Internet* family of protocols, which are the only protocols that we'll examine in this paper.

Sockets, protocol families, and interfaces

There are three major layers in the 4.2BSD networking code, going from the user level towards the hardware: sockets, protocols, and interfaces. A *socket* is an endpoint for communication: you can push bits in using **write**, and you can get bits out using **read**. There are three major flavors of sockets: *datagram*, *stream* (also known as *virtual circuit*), and *raw*. Datagram sockets support connectionless communication with no guarantee of delivery. Stream sockets support connection-based communication guaranteeing that messages will be delivered uncorrupted in the order that they are sent [Tanenbaum]. Raw sockets provide user-level programs with the same interface that routines in the kernel would see. When you create a socket, you specify a flavor, a protocol family, and optionally a protocol within the family. If you don't specify a protocol, the kernel picks an appropriate protocol, based on the flavor of the socket. To actually use the socket, you must associate it with an address. The form of this address is specific to the protocol family, as we shall discuss next. You can also set option flags for the socket, for instance whether the socket can accept connection attempts from other sockets, or whether the kernel will generate debugging messages when the socket is used. Connections between sockets are created asymmetrically. Assuming a client-server model, the server process creates a *passive* socket; client processes create *active* sockets and **connect** to the passive one. The **accept** routine returns to the server a second socket that is connected to the client's active socket, and the passive socket continues to wait for new connections. The passive socket may have a queue of connections that have been made but not yet **accept**ed. Attempts to connect to a passive socket whose queue is full are dropped on the floor.

Each protocol family has its own notions of addressing, message types, and message semantics. For example, an address in the UNIX domain is simply a path name (i.e., a string). An address in the Internet domain is a collection of integers with complex interpretations. Nevertheless, each family in 4.2BSD must support operations that are protocol- and family-independent. These requirements lead to an implementation based extensively on C *casts* [Kernighan & Ritchie] and a protocol family *switch table*: each protocol family is assigned an index into an array, and each element of the array holds pointers to family-specific functions.

Once a protocol has decided to send a packet, it must send that packet to the appropriate device interface (e.g., "il0," the host's first Interlan Ethernet controller). Finding that interface is the job of the routing code, which we shall explain below. Although there is not exactly a switch table for the interfaces, the idea is the same. The protocol is given a handle that contains pointers to the routines implementing the interface's common operations.

Routing

The routing code in 4.2BSD is for the most part protocol family-independent, but it makes use of protocol family-dependent routines (via the switch table) for hashing and some address-matching. The kernel keeps a hashed set of routing entries that match destinations with interfaces. The entries contain flags telling, for example, whether the interface connects directly to the final destination (the alternative being that it connects to a gateway), or whether the "destination" refers to a specific host or to a network. The kernel first tries to find a routing entry specific to the destination host. If that fails, it tries to find a routing entry for the network that the destination is on. If that fails, it looks for a wildcard routing entry that it uses to pass the packet off to some other machine, presumably a smarter gateway. This smarter gateway can then send back a *redirect* message with the real routing information for the destination.

The routing table is maintained by a user-level routing daemon. The daemon can change the table via two *ioctl* requests on the **/dev/kmem** device (ioctl's are explained in Section 3.1). One request adds a route, the other deletes a route. Also, the daemon can exchange routing information with routing processes on other machines, for example via raw ICMP [ICMP] sockets.

mbufs

Before there were mbufs, memory allocation in the kernel was crude. You could allocate entire pages, and you could free entire pages, and that was it. Mbufs support dynamic allocation of smaller amounts of data, for example enough to hold a socket structure. Each mbuf holds a small amount (112 bytes) of data and some pointers. There are two ways to refer to data that won't fit inside the mbuf. The first way is to string mbufs into a chain. The other way is for the mbuf to point to a *cluster*, which is one or more pages containing only data (clusters are explained in more detail in Section 4.1). Clusters are especially useful for moving large amounts of data: rather than running through an mbuf chain and copying each byte individually, you change the page table entry for the destination and then increment the cluster's reference count.

When 4.2BSD boots, it allocates a small amount of memory to clusters and mbufs. If it needs additional memory, it allocates more pages using the old, crude allocator and breaks them up into mbufs. Once it has allocated this additional memory, it never gives it back.

Internet Protocols

The 4.2BSD release provided code for five protocol families (also known as *domains*): UNIX, (DARPA) Internet, IMP (for communicating with ARPAnet Interface Message Processors), PUP (Xerox Corp.), and DECnet (Digital Equipment Corp.). Most of the IPC-based "system" applications in 4.2BSD (**rlogin**, **rcp**, **rwhod**, **routed**, etc.) use the Internet domain. One reason for this bias is that the UNIX domain does not currently support communication between different machines. Also, most of the machines that Berkeley systems want to talk to speak Internet protocols.

When we critique protocol instrumentation later in this paper, we shall only consider Internet protocols. The UNIX domain is uninteresting in part because it is rarely used. It is also uninteresting because its current implementation mostly just copies mbufs from one location to the next, a task that we leave to **gprof** and **kgmon** to monitor. The PUP and DECnet code is uninteresting because it's old and rotting. In fact, the PUP code no longer even compiles at Berkeley.

There are four Internet protocols whose instrumentation we shall discuss. The Internet Protocol (IP), a low-cost datagram protocol, is the backbone of the Internet family [IP]. The User Datagram Protocol (UDP) is a more expensive datagram protocol [UDP]. The primary difference between UDP and IP is that an IP packet may be delivered damaged, whereas the UDP code throws corrupt packets away. The Transmission Control Protocol (TCP) is a byte-stream protocol based on IP (not UDP) [TCP]. The Internet Control Message Protocol (ICMP) defines control messages sent between Internet-speaking hosts [ICMP].

2.2 Existing Tools

There is really only one program that instruments the IPC subsystem: **netstat**. Much of the instrumentation in 4.2BSD is based on special counters kept by the kernel or on data structures used by the kernel for normal operation. **Netstat** is no exception to this rule. In the rest of this section, we'll present each of the ways you can use **netstat** and explain the information it presents.

netstat

Netstat's default behavior is to dump information about the sockets in the system that are associated with Internet protocols. One line might look like:

```
tcp    709    0  ucbarpa.1018    ucbmonet.login     ESTABLISHED
```

which says that the socket is associated with a TCP connection; it has 709 bytes on its input queue; it has 0 bytes on its output queue; its local address is port 1018 (decimal) on host "ucbarpa"; the other end of the TCP connection is the login server on host "ucbmonet"; and the connection is in the "ESTABLISHED" state [TCP].

Each protocol that the system knows about is listed in the file **/etc/protocols**. **Netstat** uses the **getprotoent** subroutine to go through these protocols one at a time, comparing each with a built-in list of protocols. Every time there is a match, **netstat** displays information for that protocol's *protocol control block*. In the case of UDP or TCP, the protocol control block is part of a circular linked list; each element in the list points to a socket that uses the protocol. **Netstat** just follows the list and dumps the status of each socket.

Options to **netstat** can change this behavior somewhat. The **-n** option tells **netstat** not to print host addresses or port numbers symbolically. Thus the above example might appear as:

```
tcp    709    0  128.32.0.4.1018    128.32.0.7.513    ESTABLISHED
```

The **-A** flag tells **netstat** to print the kernel address of the socket's *protocol control block* (PCB). This structure, which **netstat** thinks exists only for TCP connections, contains TCP-specific information like the state of the connection. Ordinarily, **netstat** only prints information about the sockets that are bound to specific addresses. The **-a** flag tells **netstat** also to print information about sockets that are bound to a wild-card address. These sockets are typically used by server daemons; the socket is bound to a fixed port, but the host address is a wild-card because the host may have several Internet addresses. Finally, the user may also specify *system* and *core* files, which allows **netstat** to work from crash dumps rather than the currently running system.

Hidden away in the **netstat** source code is also a **-u** option. This flag displays information for UNIX domain sockets (which don't have protocols associated with them). In this case one line might look like:

```
8053eb0c stream    0  20     0  8053e18c 0  0
```

This line says that the socket structure starts at address 0x8053eb0c ("0x" means that the number is hexadecimal), its flavor is "stream," it has 0 bytes on its input queue, it has 20 bytes on its output queue, it is not associated with any inode, it is connected to another socket whose protocol control block is 0x8053e18c, no sockets refer to it, and it is does not refer to any other socket. If the socket at the other end of the connection had a path name associated with it, that name would be displayed at the end of the line. (A line like this one might describe one end of a UNIX *pipe* [Ritchie & Thompson].)

netstat -s and other protocol information

Netstat will also give summary information for each protocol. The **-s** option tells **netstat** to print statistics for all the protocols it knows about (which in 4.2BSD are only IP, UDP, TCP, and ICMP). The **-p** option gives statistics for whatever protocol the user asks for. Many of the protocol-specific numbers are the counts of packets that the protocol code threw away because of, say, a checksum error or because the header was given as being larger than the header and data combined. The ICMP statistics also give the counts of each type of ICMP message that the host has either generated or received.

netstat -i[t], netstat -h

These two **netstat** options give device-related information. **Netstat -i** gives information about each of the interfaces that the host is configured with. On one line appear the name of the interface (e.g., "imp0"); the largest packet size that the device can handle, which is called its *Maximum Transmission Unit* (MTU); the name of the network the interface connects to (e.g., "arpanet"); the name that the host is known by on that network (e.g., "ucb-arpa"); the number of packets read in; the number of input errors; the number of packets sent out; the number of output errors; and the number of collisions. If you say **netstat -it**, you also see the value of the *interface timer*. To see information specific to one interface, for example **en0**, use the **-I** flag: "netstat -Ien0."

Not all interfaces use timers. For those that do, the timer value is decremented once per second. When the timer expires, the **if_watchdog** routine for the interface is called and the timer is reset.

If you type **netstat** *n*, then **netstat** iterates. Every *n* seconds it displays the traffic and error counts for the busiest interface (i.e., the interface with the most traffic since the system booted), and it displays the traffic and error counts summed over all interfaces in the system. The first line has the counts since boot time; subsequent lines are the counts since the previous line. The **-t** flag is ignored in this case, and **-I** replaces the busiest interface with the specified interface.

Netstat -h gives information peculiar to IMP connections. Each line given by **netstat -h** refers to a remote host that the local host is communicating with or was recently communicating with. The **Qcnt** is the number of queued messages, waiting to be sent to the IMP. **Q Address** is the address of the first message in the queue. **RFNM** is the number of *RFNM* (Request for Next Message) packets [Malis] that the local host expects to receive from the remote host. If nothing is queued and no RFNMs are outstanding, the structure is given a short time to live. If nothing happens within that time, the structure is deallocated. Thus **Timer** shows how long the structure has left to live.

An IMP will allow the local host to begin a new transmission to a remote host before receiving a RFNM for the previous message to that host. However, the IMP will only allow up to 8 outstanding RFNMs per remote host. An attempt to send a 9th message will cause the IMP to block, which prevents communication via that IMP with any other remote system, until a RFNM comes in from the host. "Timer" is actually the number of "slow" timeouts left to the structure; slow timeouts happen every 500 milliseconds.

netstat -m

The **-m** option of **netstat** reports on the memory allocated by the IPC subsystem. It gives the number of mbufs that exist, it gives the number that are in use, and it categorizes the mbufs in use. Categories include message data, socket structures, and entries in the routing table. Mbufs are sometimes used for purposes that have absolutely nothing to do with IPC, like holding the accounting information for a zombie process until the parent process *reaps* it (reaping is described in Section 5.1). This happens simply because other parts of the kernel are taking advantage of the improved service that the mbuf scheme provides. **Netstat** also gives usage numbers for clusters, it tells how much memory is being held by the networking subsystem, and it tells how many requests for memory were denied.

netstat -r[s]

Netstat -r dumps the routing tables. A line using this option might look like:

```
washington     uw-vlsi-gw     UG     1  11427     imp0
```

This says that the entry is for a network (no **H**--for "host"--flag), the **washington** network in particular. Packets to this network should be addressed to **uw-vlsi-gw**, which is a gateway (**G** flag), via the **imp0** interface. The gateway is up (**U** flag), there is one instance of someone using that route, and 11427 packets were sent using that route. If you specify **-rs**, then **netstat** instead gives a summary of the redirect messages that have been received, plus, for comparison, how many times the wildcard route was used.

2.3 Problems with the Existing Tools

In this section, we shall discuss what **netstat** does wrong and what it lacks. We feel that network instrumentation should say when problems are coming from outside as well as from within. Thus we shall also present some measurements that tell what other hosts are doing to the local machine.

netstat

Our first complaint ensues from the detail that **netstat** provides as its default. We expect that the first step in checking a system will be to get a general feeling for what the system is doing: is it using the disk heavily, how much is it paging, how many processes are sharing the CPU, what's the network traffic like. The most general invocation of **netstat** should give the most general information. If you want more specific information, then you can specifically ask for it by giving flags to **netstat**. The appropriate general information for IPC is an iterating display of network traffic. That is, we could just make the **netstat** *n* option be the default for **netstat** (including the changes we suggest below for **netstat** *n*). An added advantage of this iterating display is its consistency with **iostat**'s and **vmstat**'s interface.

The socket information that **netstat** gives now is useful for debugging distributed programs. That information should therefore still be available, but as a new option. That option would be even more useful if it displayed the socket's option flags, preferably as a hexadecimal number. (The flags are defined in **socket.h** as hexadecimal constants. If the flags were defined as octal constants, then **netstat** should print them in octal.) Also, **netstat**'s *man page* (the writeup in [UserManual] or [ProgrammerManual]) should at least give pointers to documents that describe the different states a connection could be in (e.g., [TCP]). It would be even better if the man page itself provided that information.

netstat -s and other protocol information

The most obvious hole in this area is that the **-p** option exists only in this paper and the manual page; it has not yet been implemented. The code for **-p** should be a straightforward change to the code for the **-s** option.

Our complaint about the **-s** option is that it is too detailed in one spot. Although the individual error counts for each protocol should be kept for debugging work, it would be more useful for instrumentation to lump all of the errors together into one "number of packets dropped" count, especially as most of the counts are usually zero. The total number of packets sent and received should be a part of each protocol's statistics to give an idea of how important the errors are. In addition, we propose the following additions for the protocols:

<u>IP</u>

- number of packets to be forwarded
- number of packets actually forwarded

  These numbers, combined with the number of packets read in, give an idea of how much load a host's gateway duties are causing.

- number of packets dropped because their *Time To Live* (TTL) expired [IP]

  TTL information gives some idea of network congestion, routing problems, or errors at the sending host (perhaps it is making the TTL too small). The TTL checks in 4.2BSD are done in **ip_forward** and **ip_slowtimo**.

- number of input packets that had to be assembled from fragments
- number of output packets fragmented because they were larger than the device MTU (distinguishing between forwarded packets and packets originated in the host).

  Fragmentation degrades performance, hence counts of some sort would be useful. Locally sent and forwarded packets should be distinguished because locally sent packets are easier to control; a problem with forwarded packets is harder to correct. The count of input packets tells how much work is being caused by remote hosts. Input fragmentation is dealt with in **ip_reass**. The code in **ip_output** handles output fragmentation.

TCP (per host)

- number of idle connections

  A TCP connection uses 3 mbufs at each end (1 for the socket, 1 for the Internet PCB, and 1 for the TCP PCB). This gives us three-fourths of a kilobyte for each idle connection, sitting around doing nothing, which is sufficient motivation to find out how many idle connections there are. (We present a scheme for defining "idle" connections below.)

TCP (per connection)

The following information could be kept in the connection's TCP control block, which **netstat** looks at already.

- number of useless bytes received

  The number of "useless" bytes is the number of bytes that duplicate information already received. This number gives an indication of how much of the network load is avoidable. TCP reassembly, which is where duplicate bytes are dropped, is done in **tcp_reass**.

- number of timeouts

  TCP times out on a per-connection basis. A connection that times out often is a sign of network congestion or routing problems. The code that handles TCP timers is in **tcp_timer.c**.

- number of *persist* transmissions versus the total number of transmissions

  A *persist* transmission is one that TCP can use to make sure that the other end of a connection has not gone away. Comparing the number of persist transmissions with the number of transmissions actually holding data would give an idea of how much useful work is getting done.

- whether the transmission is idle

  The persist transmission timer can notify the TCP code when a connection becomes "idle." Thus the per-system TCP information would tell us how much memory TCP connections are wasting, and the per-connection information would tell us who the culprits are.

- estimated round-trip time

  TCP estimates the round-trip time between it and the remote host. It keeps this number in the connection's TCP control block and uses it to schedule timeouts. Someone experimenting with routing algorithms (or even just hand-patching the routing table) could profit from knowing these numbers.

ICMP

- the number of IP packets sent and received.

  This number would tell how significant the count of ICMP messages is. These counts would be shared with the instrumentation for IP.

netstat -i[t], netstat -h

The interface information given by **netstat** has one outright bug and a few other problems. The bug is that **netstat** doesn't really find the busiest interface when you use **netstat** *n*. Instead it just uses the first interface that it finds.

Some of the other problems are problems of adequacy: **netstat** doesn't tell you all that you'd like to know. For example, each interface has a fixed-length queue that it puts input packets on. Bottom-level protocols (e.g., IP) take the packets off these queues for processing. If there is no room in the queue, the interface code drops incoming packets on the floor. For each interface, the kernel updates the number of dropped packets with the **IF_DROP** macro, but **netstat** doesn't display these counts. Also, **netstat** should display in hexadecimal the flags associated with an interface, which would be useful for debugging. These flags are defined in **if.h**.

The interface-related code also has problems with the numbers it does display. One problem is that the first values displayed by **netstat** *n* (the number of packets read and written since boot time) are totals, rather than averages. Perhaps this is to be consistent with the displayed error counts (which would likely be zero if displayed as averages). Nevertheless, it would be much more intuitive to start out showing the average traffic, which is what **vmstat** and **iostat** do. This lets the user directly compare the next lines from **netstat** with the first set of values, to see whether the load on the system is unusually low or high. One solution is to display just traffic averages, and to use a separate option to display total error and traffic counts.

The "collisions" count that **netstat** displays for the IMP interface is thoroughly warped. The interface code wants to talk to the IMP using a specific "new" format; if it receives a message in some other format, it chalks it up as a collision on the IMP interface. It also counts messages with unrecognized or unused types as collisions. While it makes sense to record these errors, and the "collisions" count would otherwise go unused for IMPs, this practice is misleading. Instead, each interface should have one or more counters for special device-specific errors. For CSMA devices like the Ethernet, this count would be the number of collisions; for IMPs it would be something else. Thus in this case we are not proposing any real change to the structure of the instrumentation, just some textual changes to make life less confusing.

An important problem with all instrumentation code in 4.2BSD is how it schedules iterations. Rather than actually collecting and displaying numbers every, say, five seconds, **netstat** collects numbers, displays them, and then tells the kernel "call me back in five seconds." This strategy makes the collection period longer than 5 seconds: it equals 5 seconds plus however long it takes to collect and display the numbers. We shall discuss this problem further in Section 4.3.

Finally, hardware problems with an interface can generate spurious interrupts and bog down the system. One symptom of this problem is a large number of errors on the interface within a short time. However, if nobody is using the machine, the problem may go unnoticed. In the interest of finding and fixing hardware problems as soon as possible, the interface should log cases of unusually large error rates.

We can detect large error rates by keeping a second error counter that is decremented at a fixed rate. If the counter ever gets above some threshold, then interface code should log a message and reset the counter. Finding the right threshold and the right rate to decrement the counter will take some experimentation, though.

netstat -m

All of the problems with the **-m** option are cases where **netstat** doesn't tell us as much as we'd like to know. Part of the problem is that the kernel doesn't provide enough information. One thing that **netstat** could do for itself, though, is look for cluster leaks. That is, **netstat** should check that the number of free clusters plus the sum of the cluster reference counts equals the number of clusters that exist.

One place where we would like more information from the kernel is in mbuf allocation. There are two cases in which **m_clalloc** will fail to allocate more memory for mbufs. The first case is when there is simply no more core available (by "core" we mean physical memory of any sort). The second case results from the resource management scheme used inside the kernel. Allocatable resources like mbufs and swap space are described by a finite-sized **map** structure. This setup typically bounds the number of objects that can be allocated, and fragmentation in the map can even force the kernel to throw resources away. In the case of mbufs, at most **(NMBCLUSTERS - 1) * CLPAGES** pages can be allocated for mbufs and clusters, independent of the available physical memory. Thus when the mbuf code fails to allocate more memory, it should record whether the failure happened because the map is too small or because there is no core available. If the mbuf map (**mbmap**) in a system is too small, it can be grown by changing the value of **nmbclusters** (in **param.c**) and rebuilding the kernel. On the other hand, there is little point in recording unsuccessful attempts to allocate clusters. When the kernel boots it allocates exactly 32 kilobytes for clusters; it never allocates additional memory for them. We would prefer that the amount of memory for clusters be tunable, but until that happens, knowing how often a cluster allocation fails won't be of much use.

A **map** structure is implemented in 4.2BSD as a list of free resources, using a starting location and size to describe each free chunk. Thus if the resource becomes too fragmented, the resource manager will need more entries in the list than it can hold. If this happens, it throws an entry--and hence a chunk of resources--away and logs a message telling which map got chopped. Two other maps that might get fragmented are **swapmap** and **argmap**. The size of these maps, like the size of **mbmap**, is tunable. The configuration parameter **maxusers** controls how many entries are in **swapmap**, and the symbolic constant **ARGMAPSIZE** (in **map.h**) controls how many entries are in **argmap**.

As we mentioned previously, some non-IPC routines allocate mbufs for their own purposes. In fact, some device drivers even take memory from the IPC pool without registering the memory either as mbufs or as clusters. The problem here is that the allocator (**m_clalloc**) doesn't keep track of this allocated memory. In 4.2BSD this slip isn't too much of a problem: if **netstat** declares that a system has allocated all **nmbclusters** worth of memory, but that mbufs and clusters account for less than **nmbclusters**, we know where to look for core leaks. But not recording who is allocating memory from the mbuf map is sloppy programming, which might return to haunt us some day.

netstat -r

Our only real complaint about **netstat -r** is that it could be made easier to use. One count that we'd *like* to see is how often the system successfully uses the wildcard route. Unfortunately, the current setup prevents us from getting this number. If a system uses the wildcard entry, the "smart" gateway that it forwards the packet to isn't required to say whether it can forward the packet, nor is it required to send back a redirect message. Thus the system will not always know whether it was trying to send the packet to a non-existent address, or whether it just didn't know how to get to that address. We can get some idea of how often the address is bad based on the ICMP messages that the system gets back from the gateway. Unfortunately, this would tie us to the Internet family for routing instrumentation. Because we want to keep domain independence for instrumenting the (domain-independent) routing code, we'll just forget about getting this information.

The problem with the interface to **-r** is that it doesn't let you specify an "interesting" route. This forces you to use **grep** ("get regular expression") when looking for a specific route in a large table. One drawback to using **grep** is that it causes extra context switches: **netstat** fills the pipeline buffer, **grep** empties it, **netstat** fills it again, and so on. Also, buffering by C's I/O library usually makes you wait until **netstat** has gone through the entire routing table before you can see anything. Instead, **netstat** should recognize commands of the form **netstat -r** *foo*, where *foo* is the name of the host or network that you want the routing entry for.

system traffic vs. user traffic

Just as **iostat** and **vmstat** break down CPU usage into "system" time and "user" time, it would be useful to know how much network traffic is from "system" functions and how much is from "user" functions. The problem, though, is how to define "system" functions so that they can be measured efficiently. The most obvious approach is to say that traffic through sockets owned by **root** or **daemon** is "system" traffic. There are two problems with this approach. The first problem is that parent and child processes can share a socket, and either process can change its user id (e.g., by **exec**'ing a *setuid* program, as explained in Section 5.1), so there is no clear ownership of sockets. There is also the problem of deciding how to classify traffic in which one end is root and the other end is just some random user. The second problem is that many programs in 4.2BSD provide "user" functions but run as **root** (the super-user) for security reasons (e.g., **rcp**). A variant of this approach is to classify the socket based on the program, rather than the user. This also has the ownership problem. Another approach is to classify the traffic based on the Internet port number at one or both ends of a connection (or datagram message). The problems here are relying on the Internet naming scheme and having to make arbitrary decisions about what is "system" traffic and what is "user" traffic.

Nevertheless, we can use a variant of this last approach to get an idea of who is using the network. If we distinguish between "instrumentation" traffic [Kupfer], "mail" traffic, "clock synchronization" traffic [Gusella & Zatti], "file service" traffic [Hunter], **rwhod** traffic, and other services, we avoid the problem of defining "system" traffic, we avoid boxing ourselves into one protocol family, and we still have a good picture of how the network is being used.

Each protocol family could implement this technique by keeping a list of distinguished addresses. Associated with each distinguished address would be a pointer to the traffic counter, which could be copied into the **socket** structure. (The counter could be for the entire system or just for that protocol family.) When bytes pass through the socket, the kernel would increment that counter. For example, when an Internet stream socket **connects** or **binds** to such a distinguished address, the pointer could be filled in then. An Internet datagram socket, however, would have a **NULL** pointer in the socket structure; **sosend** and **soreceive** would use a new protocol-family request to ask the Internet code for the appropriate counter.

profiling distributed programs

One powerful UNIX tool is the *profiler*, which tells how much time a program is spending in what parts of its code [Graham *et al*]. However, UNIX profiling only works while the process is running; time spent in the "sleep" state or waiting to run is not counted. Suppose you have a program that uses a remote service by means of Remote Procedure Calls (RPC) [Birrell & Nelson] [Cooper], and suppose that the client typically waits a long time before receiving results from the server. This delay will not show up in the profile, and you'll be left wondering why on earth your program is so slow. One possible way around this problem is to take over a machine and profile both the client and the server on it. The clock time required to run the client, less the total CPU time consumed by the two processes, is the time attributable to communication delays, such as time in the Ethernet driver or time spent accepting a packet. Of course, this solution is not practical even now at sites with few machines, and it will be even less practical if many people are each developing such programs.

Another possible solution is for the kernel to record the time required to complete certain IPC-related events, like sending and receiving an RPC packet, and make that information available to the profiler. This is something of a hack to the profiling system, but its scope would be limited to selected IPC routines. Also, it may prove to be very useful despite load variations at the server. It would provide information about delays caused by the remote machine's kernel, network congestion, and gateways. This information, which the first method cannot always supply, would at least distinguish local delays from remote delays. This approach is similar to that of the Distributed Programs Monitor [Macrander], but it differs in that it is integrated, for better or for worse, with the existing Berkeley UNIX profiling tools.

For example, to record the time required to open a stream connection, **soisconnecting** could put a timestamp in the **socket** structure. Using that timestamp, **soisconnected** could add to a counter in the **socket** structure the time needed to complete the connection (also incrementing a count of how many connetions were made). When the process exits, it can collect in one system call the profiling information for that socket.

miscellanea

As we described earlier, if a passive socket's connection queue is full, new connection attempts are rejected in **sonewconn**. The active host is left to time out and assume that the passive host is down. In 4.2BSD the queue is limited to only 5 connections (**SOMAXCONN**, defined in **socket.h**). This small size pretty much forces connection-oriented servers into an implementation style in which one process **accepts** a new connection and then **forks** off a child process to handle it, even if the service is fairly trivial (**fork** is described in Section 5.1). Because servers may need TCP for its guaranteed delivery, not because they need a virtual circuit, we feel that a server should have the option of **accepting** and processing requests one at a time, rather than **forking** off new processes to do the servicing. The small queue size also means that an errant host could block all other attempts to use a service by flooding the server with connection requests. Thus, when 4.2BSD can't put a connection on the queue, it should log that fact and identify the connections that are already in the queue.

Also, just as a system should log cases where a socket is flooded with connection attempts, it should log cases where a host is flooded with Address Resolution Protocol requests [Plummer]. The system should also record calls to a protocol's **pr_drain** routine, which is called when the kernel is running low on memory and wants to reclaim space used for, say, caches.

One chore that *internetworks* (interconnected networks) brought with them is network configuration: deciding what hardware to use, how many networks to talk to, and where to put gateways. Knowing how many bytes two hosts sent each other, similar to the information that **netstat -r** provides, would certainly aid in this planning.

One way to implement this idea would be to periodically examine the routing table, which already records the number of packets sent along each route. The problem with this scheme is that most of the routing entries will be for networks, and we want host-specific numbers. Thus it looks like the kernel will need another table to record this per-host traffic. To avoid wasting too much space, the kernel could periodically throw out entries whose traffic count is under some threshold, or a user process could periodically flush the kernel table after updating a disk file with it.

A useful tool for program debugging would record network traffic destined for or sent from the machine on which the process is being debugged. In fact, the Distributed Programs Monitor can already do this for us. In addition, we want the option of recording messages as an eavesdropping third party, so that recording the packets doesn't disturb the system being debugged. The **PupWatch** tool at Xerox's Palo Alto Research Center does just that for their systems. We would like a similar UNIX-based tool to monitor Ethernet traffic here at Berkeley. Unfortunately, such tools require help from the hardware: the network interface hardware must be willing to accept packets that aren't bound for it [Schoch & Hupp].

Our final suggestion is that **ps** optionally display for each process the PCB addresses for the process's sockets. First of all, this would be a big help for debugging. Also, it makes no sense to flag idle or otherwise troublesome sockets if there is no way to trace them back to the program that is actually causing the problem.

To find all the sockets, **ps** need merely run through the array of file descriptors in the process's u. area. **NULL** entries can be ignored, and a non-**NULL** entry will point to an entry in the file table. This entry will have a flag telling whether it's a socket. If it is a socket, the **f_data** field will point to the socket structure, which will have the address of the PCB.

3. I/O and the File System

3.1 How it Works

Since its inception UNIX has supported an elegant I/O and file system [Ritchie]. A major change that 4.2BSD brought is a new file system, one with both functional and performance enhancements. We shall first describe in general the UNIX file system, then we'll describe the notion of mounted filesystems, and finally we'll describe the changes introduced in 4.2BSD.

regular files and special files

Each UNIX file is associated with a descriptor structure, called an *inode* in UNIX jargon. This structure gives common information like the owner of the file, when it was last accessed, and who can read it. It also tells what device the file is associated with and the type of the file. *Regular* files are what you normally think of as files: a collection of bytes, usually on a disk. UNIX also supports the notion of *special* files, which are typically devices like printers, terminals, or tape drives. Files are named by giving a path in a directory tree, and you can't deduce the file's type from its name. Thus, as with IPC, the implementation is based on a switch table, this time using the device number as an index into the table.

There are two types of special files (devices): *block* special files and *character* special files. Block devices are named for an organization based on randomly addressable blocks. They share a common pool of I/O buffers, and they have a highly structured interface that is well-suited for disk I/O. A character device is anything that isn't a block device.

A process accesses a file through a *file descriptor*, which is simply an index into a per-process array of pointers. These pointers point to entries in the *file table.* UNIX lets processes share a file, down to the point of sharing the byte offset in the file. UNIX also lets processes share a file such that their accesses are independent. This is done by having each entry in the file table hold an offset value and point to an entry in the *inode table.* There is exactly one entry in the inode table for every file that is being referenced. Processes that share the file and the offset use the same entry in the file table. Processes that share the file but have different offsets have different entries in the file table. But each of the file table entries refers to the same entry in the inode table, as shown in Figure 3.1.

The 4.2BSD kernel allocates pages for the the I/O buffers separate from the buffer headers. When the system allocates a buffer, it checks that the number of bytes associated with the header is the number desired. If not, it reallocates pages for that buffer to get the right number of bytes. Three values control the allocation of buffer pages and headers. **Bufpages** is the number of *clusters* allocated for buffer pages (clusters are described in Section 4.1). If **bufpages** is zero at boot time, the kernel computes it as a fraction (five to ten percent) of the physical memory in the system. Similarly, **nbuf** is the number of buffer headers. If **nbuf** is zero at boot time, the kernel allocates one half as many headers as there are clusters, with a minimum of 16 headers. **SYSPTSIZE** is a symbolic constant that controls the size of the kernel-space page table. If **SYSPTSIZE** is so small that there aren't enough page table entries for all the I/O buffers, **nbuf** is reduced so that there is enough room. Similarly, **bufpages** is reduced if it is greater than the number of clusters that the buffer headers could possibly refer to at once.

mounted filesystems

In UNIX systems a disk pack is typically broken up into two or more disjoint *partitions*. UNIX lets you associate these partitions with subtrees in the directory hierarchy. You do this by *mounting* the partition ("filesystem") on a "leaf" directory in the existing tree. Future references through that directory are shunted over to the associated filesystem [Thompson]. This scheme is very convenient for such purposes as moving disk packs between machines (e.g., if one machine has a power supply

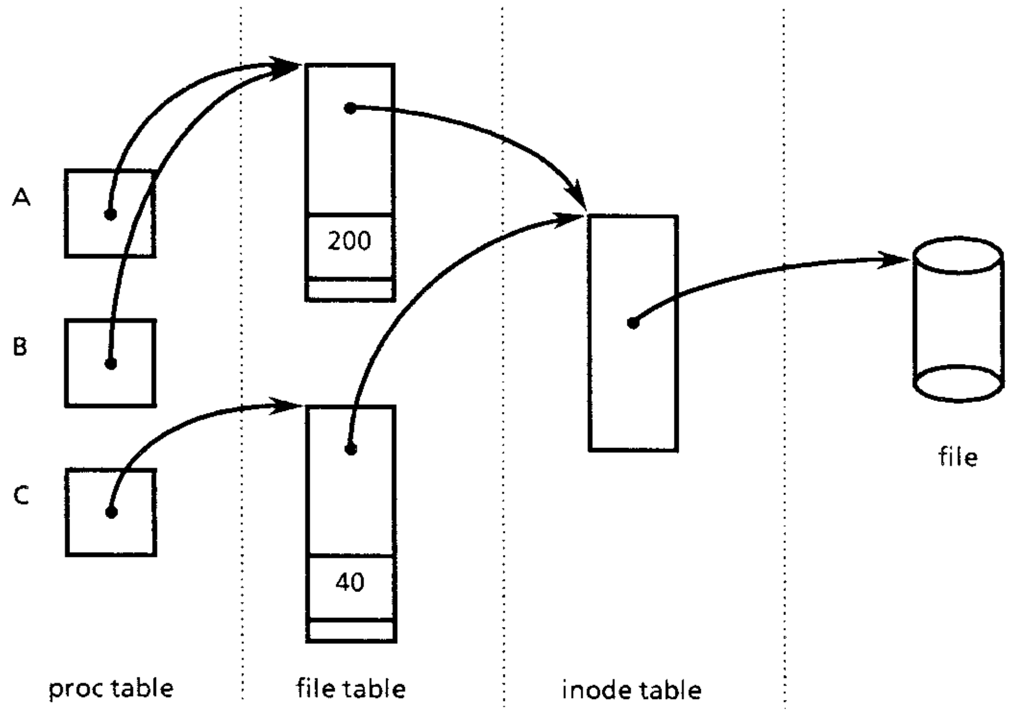

Fig. 3.1: The **file** and **inode** tables. Processes A and B share the same **file** table entry; the next one to access the file will be at offset 200. Process C has a different **file** table entry; it will accesse the file at offset 40. All three processes use the same entry in the **inode** table.

failure). Files must exist wholly within a filesystem. This limits the size of files, though this is not serious in practice, and it provides a cheap way to control and organize disk usage.

4.2BSD enhancements

The original UNIX file system was slow [McKusick *et al* 84]. One reason was that it transferred blocks of 512 bytes, which meant that per-transfer overhead had a noticeable effect. Another problem was random block layout. When files were freed, their disk blocks were merely placed at the end of a free-list. New blocks for files were merely taken off the head of the list. After a while, the partition would become a complete jumble. Sequential blocks in a file would be on different tracks, forcing a seek for every transfer, and blocks on the same track would be randomly scattered among the different sectors.

The 4.2BSD file system fixed these problems. The too-small block size problem was fixed by increasing the block size, typically to 4096 bytes. To avoid internal fragmentation of the disk, 4.2BSD introduced the notion of a *fragment*, which, for a 4K block size, might be 512 bytes or 1024 bytes. Every file consists of some number of blocks followed by some fraction of a block (some integral number of fragments). If the file grows, the disk block allocator tries to expand the chunk at the end of the file. If that fails, it allocates a new, bigger chunk, copies the old chunk over, and then frees the old chunk.

The fix for the layout problem is a smarter disk block allocator. It tries to arrange blocks so that seeks are few and far between, and it arranges blocks so that multi-sector transfers can be done with a minimum of rotational delay (by "sector" we mean the basic block of the disk, which is usually smaller than what the kernel thinks the block size is). The 4.2BSD file system reduces seeks by dividing each filesystem into *cylinder groups*. The allocator tries to put all of a file's blocks within one cylinder group until the file reaches a certain size. Then it looks for a new, under-used cylinder group to continue the file in. For this scheme to work in practice, though, the file system must keep **minfree** percent of the disk free (10% at Berkeley). That is, if a partition is down to **minfree** percent free, the allocator refuses to allocate any blocks in it (unless the allocating process is owned by **root**).

All of these enhancements were designed to be as flexible as possible. Thus all of the controlling values are filesystem-specific (not system-specific) parameters. In the worst case (e.g., changing the fragment size), you can change the parameters for the filesystem by dumping it to tape, regenerating a virgin filesystem with the new parameters, and then reloading the old files from tape. Other parameters, such as the required minimum free space on disk or the expected delay to process a disk interrupt, can be changed without rebuilding the filesystem.

System configuration in 4.2BSD was also designed to be flexible. The descriptions that you put in a **config** file only say that the device *might* be attached to the system. When the system boots, it goes through a phase called *auto-configuration*. The auto-configuration code (e.g., in **mbaconfig**) probes each device listed in the **config** file, checking whether the device really is there. If there is no response to the probe, the device is ignored.

The 4.2BSD release also introduced some functional enhancements to UNIX. Disk quotas allow a manager to limit the number of files and the number of blocks that a user may create. Shared and exclusive (advisory) locks let processes coordinate their accesses to files (e.g., for database work).

3.2 Existing Tools

There are many more tools for the I/O and file system part of 4.2BSD than there are for networking. **Iostat** gives general information about the system's disks, **pstat** displays portions of various I/O-related kernel tables, **dumpfs** dumps a filesystem's parameters, and **vmstat** gives information about interrupts.

iostat and df

Iostat doesn't have any real options: you can specify how often it will iterate, and you can specify how many times it will iterate, but it always gives you the same numbers. First comes the average number of characters read by terminals ("ttys" in UNIX jargon), then comes the average number of characters sent out by terminals. This includes traffic both from "real" terminals and from *pseudo-ttys* (called "ptys" in [ProgrammerManual]). Then, for up to **DK_NDRIVE** disks, **iostat** gives the following information: average number of kilobytes transferred per second, average number of transfers per second, and average number of milliseconds per seek. Finally, the percent of time that the CPU spent in user mode, executing *nice*'d processes in user mode, in system mode, and idle are printed. ("Nice" is explained in Section 5.1.) The first line given by **iostat** contains averages since the system booted. Thereafter the line contains averages since the previous line was printed. For each drive, 4.2BSD keeps the number of clock ticks the drive is busy, the number of seeks done by the drive, the number of transfers done by the disk, the number of (16-bit) words transferred, and the disk's transfer rate. The average seek time is computed by calculating how much time the disk spent transferring data, subtracting that from the time the disk was busy, and dividing by the number of seeks done by the disk.

Df tells how full the disks are. For each filesystem it gives the name of the associated disk partition (a special device, e.g., **/dev/hp0a**), the number of kilobytes in the filesystem, the number of kilobytes in use, the number of kilobytes available, and the percentage of the filesystem that is in use. The number of kilobytes available and the percentage in use are calculated after taking away the 10% minimum allocation described previously in "4.2BSD enhancements." Thus **df** occasionally reports that a partition is, say, 102% full. The **-i** option tells **df** to include the number of inodes in use, the number of available inodes, and the percentage of inodes in use.

pstat

Pstat is not so much an instrumentation tool as it is a debugging tool (e.g., for analyzing crash dumps). Nevertheless, its **-s** option *is* useful for instrumentation. As long as we're looking at that option (in Section 4.2), we might as well look at the rest of the program. **Pstat** has three options that we are interested in here (there is no default to **pstat**). Two of these options are related to the file system, and one is related to terminals. **Pstat -i** dumps the in-core inode table. It gives data such as the address of each entry, the inode's flags, the number of references in the file table to the entry, and the user ID of the owner. The flags are printed symbolically: each bit has a letter associated with it that is printed if that bit is on. For example, **A** means that the file system needs to update the file's last-access time, and **E** means that a process has an exclusive lock on the file.

Pstat -f provides similar information about the file table. An entry in the file table can refer to objects other than files. In particular, it can also refer to a socket. If the entry is a file, **DATA** is the address of the file's entry in the inode table. Otherwise it is the address of the socket structure. **MSG** is the number of in-flight messages that refer to that file-descriptor via an access-rights mechanism (implemented only in the UNIX domain).

The **-t** option of **pstat** prints information about all the terminals in the system, including the console and pseudo-terminals. **Pstat** arranges terminals according to terminal controller (e.g., DZ, DH). It gives information like the size of the terminal's input and output queues, its flags (which are defined with the terminal ioctls in **ioctl.h**), its state, the address of some controller-specific values, and the column that the kernel thinks the cursor is at (going from 0 through 255).

dumpfs

Dumpfs tells you more than you'll ever want to know about a filesystem. The first set of numbers are stored in the filesystem's *super block*; defined in **fs.h**, they are mostly filesystem parameters and other numbers used by the disk allocator. Next comes a map showing how blocks are laid out on disk and summary information about each cylinder group in the filesystem (e.g., number of free blocks, number of free inodes). Finally **dumpfs** dumps information about each cylinder group, down to the point of listing which inodes are taken and which fragments are free.

vmstat -i, vmstat -s

Vmstat, in addition to providing information related purely to virtual memory, also gives information about interrupts. The **-i** option tells **vmstat** to dump an array of interrupt counters, giving for each device the total number of interrupts and the average number of interrupts since the system booted. The **-s** flag causes **vmstat** to dump the **sum** structure, which is a counter of events mostly related to virtual memory. The **sum** structure also includes the number of device interrupts, the number of software interrupts, and the number of interrupts from DZ terminal controllers that **locore** (the assembly language code at the very bottom of 4.2BSD) uses to simulate a DMA interface. (That is, the DZ driver, which is written in C, sees a DMA interface thanks to the code in **locore**.) The **sum** structure also includes the number of traps and system calls, which can be compared against the number of interrupts. The total number of interrupts given by **-i** should equal the number of device interrupts given by **-s** plus the number of pseudo-DMA interrupts given by **-s**.

Vmstat -s also gives statistics on the **namei** cache, which was introduced after 4.2BSD was released. **Namei** is the kernel routine that converts a path name (a string) to a pointer to an in-core inode structure. A typical 4.2BSD system might spend 11% of all its cycles doing **namei** lookups [Leffler *et al* 84]. For this reason **namei** now caches recent translations, and it keeps information about the cache's effectiveness.

3.3 Problems with the Existing Tools

iostat and df

There are a few things that we'd like to get from **iostat** but don't. First, **iostat** only gives per-disk information. One benefit of mounted filesystems is that you can spread users' files across all of the disks in the system, and you can shuffle them around, for the most part transparently. This helps avoid having one disk being too much of a bottleneck. Of course, after a while a system's user population can change, and what was once a well-tuned arrangement can become out-of-balance. If

iostat gave per-partition traffic, rather than per-disk, the job of retuning the filesystems (or even just knowing if retuning would help) would be easier.

The header for an I/O buffer could contain a pointer to the traffic counters for the appropriate partition. The pointer would get set by **getblk** or **geteblk**. The counters would get updated either by the disk driver code or by the buffer I/O functions that call the disk's *strategy* routine. The counters would also need information (e.g., device number) telling what disk partition they are associated with. If the counters were kept in a linked list, with the head of the list in a well-known location, **iostat** could trace down the list to display the per-partition numbers.

A major hole in disk-related instrumentation is that **DK_NDRIVE** (the number of instrumented drives) is defined in **dk.h** as always being 4. Furthermore, the drives instrumented are simply the first four that the auto-configuration code finds, they aren't even the four busiest drives. This limit may have been placed to reduce the time that 4.2BSD spends doing instrumentation, but there's no point in halfway measures. All disks in the system (well, all partitions in the system) should be instrumented.

The 4.2BSD kernel finds out what devices are available after it has set up physical memory. Perhaps configuration could be done before physical memory is set, in which case the counters for each partition could be allocated with the process table, inode table, I/O buffers, and so forth. If not, then either the counters could be kept in mbufs, or **startup** could use some scheme to find out before configuration how many counters to allocate.

Rather than give for each disk the average number of requests per second, **iostat** should usually give the percent utilization (i.e., the percent of the time that the disk was marked busy). In general we are more interested in knowing that a disk was busy, say, 50% of the time, than in knowing exactly how many requests were made per second. The number of requests per second and the number of kilobytes transferred per second should be obtainable via a separate option.

Although **vmstat** already gives us the statistics on the **namei** cache, it really makes more sense for those numbers to be a part of **iostat**. Also, as a tuning aid, **iostat** should give statistics on the usefulness of the buffer I/O cache, such as the number of hits, the number of misses, and the size of the cache. The 4.2BSD start-up code writes the size of the buffer I/O cache and the number of buffers on the console. This information is therefore available in **/usr/adm/messages.** If **SYSPTSIZE** does limit the size of the buffer cache, a message noting the limit will also be in **/usr/adm/messages**. However, as it usually takes a bit of poking through the file to find these messages, **iostat** should include this information in with the buffer cache statistics.

pstat

Pstat has three problems. The symptom of the first problem is that it doesn't recognize all the possible flags for inodes or all the possible state bits for terminals. This hole is easily patched. What's more important is **pstat**'s attempt to print that information symbolically. First of all, using one character per flag takes up a lot of room on the screen if there are a lot of flags. Second, **pstat** quickly runs out of single-letter mnemonics for the bits (e.g., "Z" in **pstat -i** means that someone is waiting for an exclusive lock on the file). Finally, every time a new flag bit is defined, someone has to fix **pstat** to recognize that flag. A better approach is for **pstat** just to print the flags in hexadecimal or octal; the manual page for **pstat** can tell what each of the flags means. Thus **pstat** will print complete information even when new flags are added (unless another flags word is begun). It would be naive to say that the manual page would also track the new flags, but at least knowledgeable users could look in the C header file that defines the flags.

The second problem in **pstat** is related to file offsets. The offset field in a file table entry is defined as an **off_t**, which on a VAX is a 32-bit signed integer. First of all, if a file is larger than approximately 2.1 gigabytes, its offset will be treated as a negative number (which means that 4.2BSD may act strange when dealing with files that size or larger). The **dofile** function in **pstat** deals with this problem by printing "negative" offsets in hexadecimal. This whole mess would be a lot cleaner if **off_t** were simply an unsigned integer. **Pstat** would then not need to test for negative offsets, and the 4.2BSD kernel would (probably) be more robust. Similarly, the **t_col** field of a **tty** structure, which tells what column the terminal driver thinks the cursor is at, should be an unsigned **char**, not just a **char**.

The third problem is that **pstat -t** assumes it is running in a world of Digital Equipment Corp. hardware (DZ, DH, and DMF terminal controllers). Switching to a different vendor should only involve changing one table inside of **pstat**, not rewriting a moderate amount of C code. The table would simply list for each controller its name (as a printable string), and where to find information for that controller.

dumpfs

One small complaint about **dumpfs**, and then we'll get to the important stuff. The small complaint is the way **dumpfs** displays its information. There's something disconcerting about being presented with a morass of numbers and cryptic abbreviations like "ipg" and "cs[].cs_(nbfree,ndir,nifree,nffree)." These fields should read more like "inodes per group" and "summary information per cylinder group (blocks free, directories, inodes free, fragments free)."

More important, **dumpfs** should check the consistency of the numbers it produces, rather than just printing them. This is faster and less error-prone than having a human check the numbers. **Dumpfs** should also compute the number of bytes per inode in the filesystem. When you create a filesystem, you tell (perhaps indirectly) **mkfs** to create an inode for every so many bytes of data. **Mkfs** converts the inode density to the number of inodes per cylinder group, which is the value stored in the filesystem super block and is the value that **dumpfs** prints right now. We should point out, though, that the inode density that **dumpfs** computes may not agree with the one used to create the filesystem. This discrepancy might exist because **mkfs** enforces a limit of **MAXIPG** inodes per cylinder group (defined in **fs.h**).

vmstat -i, vmstat -s

The biggest question about **vmstat**'s **-i** option is why it's even there. A breakdown of system interrupts doesn't logically belong with statistics about the virtual memory subsystem. It belongs with other I/O information. The interrupt information that **-s** gives is more reasonable. It lets you compare interrupts and traps caused by paging with the total number of interrupts and traps in the system. Still, the count of "pseudo-DMA DZ interrupts" has little, if anything, to do with virtual memory and should be displayed elsewhere.

The other problem with this information is that it is inherently machine-dependent. Actually, **vmstat -i** handles this problem very well. It just looks in the kernel for an array of counts and an array of names, and it displays them. **Config** puts these arrays in **ubglue.s** based on the devices that the system is configured with. However, the **sum** structure that **vmstat -s** dumps, being defined in **/sys/h/vmmeter.h** instead of somewhere in **/sys/vax**, makes the pretense of being machine-independent. On the one hand, it is probably safe to say that any machine running UNIX has both software- and hardware-initiated interrupts. On the other hand, machines not made by Digital Equipment Corp. probably don't use DZ terminal controllers. They may use pseudo-DMA for other reasons, but in that case vmstat and its manual page should refer to the count simply as "pseudo-DMA interrupts." Furthermore, other architectures (e.g., a workstation with a bit-mapped display) may find it useful to break down interrupts into categories other than what **sum** provides. At the very least, **vmstat** should be structured so as to make this machine-dependency more explicit.

4.2BSD enhancements

In light of the 4.2BSD file system enhancements, there are three types of tools that we would like to have. The first set of tools helps tell how much benefit the enhancements really provide. The second set tells whether the enhancements are really doing what they're supposed to be doing. The last set helps us administer the file system more effectively.

The original paper on the Fast File System showed that it could speed file accesses by up to a factor of ten. The price of this speedup was 800 lines of code and comments. Unfortunately, because of the way the system was implemented and tested, we don't know which enhancements had what effect on the system. One of UNIX's advantages over other operating systems is its small size and elegance. If certain code offers little improvement in functionality or performance, it should be removed. A recent study [Ousterhout *et al*] suggests that 70% of all sequential accesses in UNIX are less than 4000 bytes long, which is less than the smallest block size in 4.2BSD. So we wonder about the usefulness of 4.2BSD's smart layout policies. The code to implement them is not trivial, and the extra processing significantly slows down file growth [McKusick *et al*].

We would like to see additional information supporting the conclusions of [Ousterhout *et al*] before ripping out chunks of the file system code. However, rather than post-analyzing file system traces, as was done in that study, we can use counters in the kernel to determine how much data is transferred sequentially and how much is transferred in random-access mode.

Each entry in the file table could have a "last seek" field that would tell where the last seek on the file had moved to. When the file is opened the field would be zero. When the file is next seeked on or closed, the kernel can determine how many bytes were transferred sequentially and increment a counter. A user program could use an array of these counters to generate a histogram. For example, the first counter could tell how many sequential tranfers were less than four kilobytes long (the minimum block size in 4.2BSD). The next counter could tell how many sequential transfers were between four and eight kilobytes, and so on.

If we conclude that the layout policies are still a good idea, then we want to make sure that they are correctly implemented. For example, student projects at Berkeley in Spring 1984 [Seymour & Lob], [Opperman & Davis] revealed a bug in the allocation code that was placing sequential sectors (i.e., physical disk blocks) of a file farther apart than was ideal. These tools were modified versions of **fsck**, **clri**, and **dumpfs**. Rather than using these tools once and then throwing them away, they should be kept as options to the original programs, though perhaps no one outside of Berkeley would want to use them.

As for administration, the quota implementation in 4.2BSD provides some tools for quota management, such as a quota editor. At Berkeley, a student's quota on "ucbcory" (used by the Computer Science Division for undergraduate classwork) is based on what classes the student is taking. Rather than manually keeping track of "special" quotas so that they can be reset at the end of the semester, ucbcory's management wrote a program to flag quotas that are higher or lower than normal.

One number that would be useful for all three of the above categories is the number of times the allocation code (in **ufs_alloc.c**) could not put a block in an optimal location. If blocks are often put in sub-optimal locations, then the complexity of the 4.2BSD layout code is wasted. Someone debugging the allocator would want to know how often the allocator thought it was putting the block in the right spot and how often it thought that the block was going somewhere else. Finally, if lots of blocks are ending up in sub-optimal places, that would be a reason for increasing **minfree** (the percentage of the filesystem that 4.2BSD keeps reserved).

logging

The 4.2BSD kernel already logs many types of errors, such as device parity errors, full system tables, and full disk partitions. We have two complaints about error logging in 4.2BSD: the information is incomplete in spots, and there are problems with how the kernel writes log messages.

One place where logging is incomplete is the disk block allocator. The kernel already writes a log message when a process fails to allocate a disk block from a full filesystem. Unfortunately, the message doesn't include the ID of the process that wanted the block. Sometimes a partition is full simply because users have not cleaned up after themselves in awhile. Other times the partition fills because some program has run amok. It is for this second case that we want the process ID.

Another deficiency is that 4.2BSD does not log cases where the in-core per-user **quota** table--represented by a linked list--is full. Because 4.2BSD allocates **quota** structures based on **MAXUSERS**, making the size of the table tunable, it should log instances when the system runs out of them. Another tunable resource whose scarcity doesn't get logged are the system I/O buffers. In this case, it may not be a good idea to log each and every time that a process has to wait in **geteblk** for an empty buffer. However, it would be useful to record the number of times this happens and write a log message if the count exceeds some threshold within a given amount of time. (We discussed this "threshold" logging in section 2.3 while discussing interface errors.)

On the other hand, we are quite content that 4.2BSD doesn't log cases where a process tries to allocate more than **NOFILE** open files. There certainly are cases where a process needs more files than normal. We believe that if this error were logged, though, most of the messages would be noise--bugs in the program, rather than a real need for many open files.

The problem with the logging implementation in 4.2BSD has two consequences. The first is inefficiency. Kernel error messages in 4.2BSD are written to the system console via a special version of **printf**. Unfortunately, other processes do not run while a message is being **printf**'d. In fact, if **printf** is called at a high interrupt level, the kernel can even miss interrupts while the message prints. Thus we have seen cases at Berkeley where a lightly loaded machine was sluggish because an infinite-loop process was writing to a full file system, causing continuous console error messages. We also have seen the case where the system clock slowly fell behind because kernel **printf**'s caused the system to miss interrupts from the hardware clock. The second consequence is that there are two classes of system error messages: those generated by the kernel, and those generated by user programs via **syslogd**. One feature of **syslogd** is that it can send messages to system managers (e.g., when someone gives an invalid super-user password). In some cases we want the kernel to send messages, too (e.g., when a disk is write-locked). To correct both of these problems, the kernel's error-logging mechanism should use **syslogd**. We suggest one change, though. When **syslogd** sends a message, it scribbles on the user's terminal. This is rude, especially if the message isn't urgent. In those cases **syslogd** should send mail instead.

In fact, selected **printf** calls have already been replaced at Berkeley by calls to a new **log** routine. This solves both of our complaints. **Syslogd** gets the kernel messages by reading from the **/dev/klog** device, and the error message is no longer printed on the console.

miscellanea

As UNIX becomes a more popular system, the average sophistication of its users drops. At some point it will become (perhaps already has become) impractical to rely on users to tell management that more resources (e.g., printers, tty lines) are needed. Therefore UNIX itself should record information about contention for and utilization of resources. This information can also point to resources that are being wasted or unused, which is something users will never complain about anyway. For example, UNIX should record the average queueing delay that files see waiting for a printer. It should also record the utilization of terminals and dial-in lines, and it should tell how much of the "in-use" resources are actually being wasted by idle users. There would be little point in recording such information about tape drives, though. For one thing, instrumentation of tape drives is potentially complex (e.g., you have to account for use by other machines if the drive is multi-ported). For another thing, tapes are usually used for file archiving and backup [Kridle]. Thus relatively few UNIX users use tapes; those that do are typically sophisticated enough that they can work out contention problems for themselves.

Another useful thing to know is how much wasted space is in a filesystem. This waste comes from two sources: fragmentation and bad blocks. Fragmentation waste comes from a file's not using all of the last fragment allocated to it. This waste can be tuned by changing the filesystem's block and fragment sizes. The number of bad blocks can be "tuned" by getting a new disk. A program like **fsck** can dig up fragmentation information while doing its normal disk-checking duties. The **bad144** program already gives information about bad-sector forwarding, though there is no foolproof way to find sectors marked bad by **badsect**. Because the number of bad sectors is probably small, we shall ignore it and only worry about fragmentation.

Our third request is that **ps** display for each process the number of disk operations done by the process (not including paging or swapping). This information, along with paging information already given by **ps**, would help us track down offending processes when the disks are heavily loaded. The display should include the number of **sync** and **fsync** calls. These calls force all buffers (**sync**) or one file's buffers (**fsync**) out to disk if they are dirty, thereby synchronizing the buffer cache with the disk. If used too often, these calls can cause a heavy load on the system.

4. Virtual Memory

4.1 How it Works

In this section we shall explain the virtual memory subsystem in 4.2BSD. Some of this code is acknowledged as being VAX-specific--it is kept in **/sys/vax**. The rest of the code is supposedly machine-independent and is kept in **/sys/sys**. However, the overall design, especially for paging, has a heavy VAX color. Our presentation will not distinguish between the VAX-specific code and the machine-independent code. Also, our description will only cover shared-text, demand-paged programs, not older formats.

First we shall describe the virtual address space seen by each user process. Then we'll describe the way 4.2BSD allocates physical memory. Third, we shall describe how it does paging and swapping. Finally, we'll describe the 4.2BSD swap device abstraction.

the virtual address space

Each user process in 4.2BSD sees three segments: text, data, and stack. The text segment holds a read-only copy of the process's code. Each program being run is described by an entry in the *text table*. Processes sharing a text segment (i.e., running the same program) share an entry in the text table. The data segment is read-write data owned only by that process. It begins after the text segment and grows towards the high end of memory. The stack segment is also read-write and is owned only by that process. It begins at the high end of memory and grows towards the data segment.

allocation of physical memory

The bootstrap program loads the 4.2BSD kernel into the beginning of physical memory. As shown in Figure 4.1, the kernel start-up code allocates adjacent physical memory for tables whose size is configuration-dependent (e.g., the inode table and the file table), buffer headers, and buffer pages. The start-up code also allocates a message buffer at the high end of physical memory; the kernel uses this buffer for error logging. The remaining physical memory is available for use by user programs, though the kernel may still allocate pieces of it (e.g., for mbufs). The kernel manages this remaining memory using the **cmap** ("core map") structure. Rather than allocating it in terms of pages, 4.2BSD deals with *clusters*. That is, each entry in **cmap** refers to a cluster of adjacent page frames, not a single frame. The symbolic constant **CLSIZE** tells how many pages are in a cluster; **CLBYTES** tells how many bytes are in a cluster.

paging and swapping

Both pageouts and swapping are handled by "system" processes. These processes are created when the system boots; they run entirely in the kernel. The paging and swapping policy depends on 4 parameters: **lotsfree**, **desfree**, **minfree**, and **maxpgio**. If there are more than **lotsfree** pages available, then the pager doesn't run at all. Otherwise, the pager scans pages using a "clock" algorithm. The farther the system is from **lotsfree**, the faster the pager scans.

The conditions for swapping are more complicated. In general, the swapper will "get desperate" (i.e., look for a process to swap out) if all three of the following conditions are satisfied:

- on the average there are at least two processes swapped in
- the paging rate is greater than **maxpgio** operations per second or there are less than **minfree** pages available
- both the 5-second and 30-second averages of available pages are below **desfree**

If the swapper isn't desperate, it takes a leisurely stroll through the process table, looking for swapped-out processes that are runnable, and looking for processes to swap out. In this case,

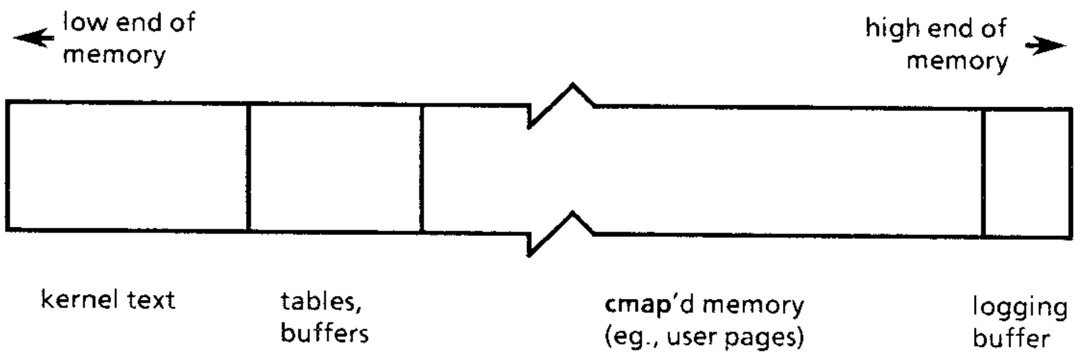

Fig. 4.1: allocation of physical memory.

processes are swapped out if they are "idle" (i.e., have been sleeping or stopped for more than **maxslp** seconds) and there are less than **desfree** pages available. If the swapper is desperate or if it has found a process to swap in, it again looks for an idle process to swap out (but is less choosy than the first time). If the swapper is desperate but can't find an idle process, it just picks the biggest available process to swap out. If the swapper wants to swap a process in (but isn't desperate) and can't find an idle one, it tries to be fair by swapping out a large process that has been swapped in "for a reasonable time."

If the swapper can't find an idle process to swap out, it finds the **nbig** processes taking up the most physical memory (**nbig** is defined in **vm_sched.c**). If the swapper wants to bring a process in, it looks among the **nbig** processes for the one that has been in core the longest. That process is swapped out only if it has been in core for more than **maxslp** seconds (thereby avoiding thrashing).

The clock-style paging in 4.2BSD is complicated by the VAX's lack of a *use bit* for each page frame. Instead, 4.2BSD uses the frame's *valid bit* to simulate a use bit. When a page fault occurs, if the page table entry points to the frame holding the page, the kernel turns the hardware valid bit on and returns from the trap. This is called a *reclaim* (because it satisfies the fault without resorting to a disk read). The pager, using the 2-handed clock algorithm described in [McKusick *et al* 85], turns off the hardware valid bit when the first hand reaches the page. If the hardware valid bit is still off when the second hand reaches the page, the page is freed, and it is written to disk if dirty. Freeing the page means putting the page frame in a free list, but leaving the page table entry pointing to the frame. This scheme allows the process to reclaim the page if it faults on it before some other process has grabbed the frame. A third way to reclaim a page is when some other process had owned it, for example when a process faults on a text page for a recently-compiled program.

A page is always marked dirty the first time it enters memory. When a dirty page is paged out, it is copied to the process's swap area, which the kernel allocated for the process when it **exec**'d the program. (**Exec** is explained in Section 5.1.) Thus when a process is swapped out, only its dirty pages must be written to disk. When a process swaps back in, it pages itself in on demand. The paging code tries to read or write adjacent clusters with a single disk request. Unneeded pages brought in this way are put on the free list in the hope that they will be used soon. This technique is called *klustering* (with a "k" to distinguish it from "clustering" with a "c"); the pages read or written at once are called a *kluster*.

When a page table is first set up, its entries are all marked *fill on demand.* Pages containing initialized data or text fill on demand from the executable file. Pages containing uninitialized data are marked *zero-fill on demand* ("zfod"). So if **pagein** doesn't find the page in core, it has three choices. If the page is not fill on demand, **pagein** brings it in from the swap device. If the page is zero-fill on demand, **pagein** allocates a frame, marks it dirty, and zeroes it. Otherwise, **pagein** allocates a frame, marks it dirty, and brings the page in from the file system.

Unless the system has little physical memory (under 2 megabytes), **lotsfree** is (equivalent to) 512 kilobytes, **desfree** is 200 kilobytes, and **minfree** is 64 kilobytes. Otherwise, these parameters are calculated as fractions of the available memory (i.e., after taking away space for the kernel, I/O buffers, and tables). **Maxpgio** is computed based on the number of disks with swap partitions. If, however, these parameters are patched to be non-zero (as described in Section 1.3), they are left at the patched value when the system boots.

The plan for 4.2BSD allows paging to take up two-thirds of the disk requests (assuming one request per disk revolution) as long as there is only one disk doing paging. As soon as there is more than one disk for paging, though, **maxpgio** is raised to allow for each disk one paging operation per revolution of the disk. Or at least that's the intention. In reality, **vminit** computes **maxpgio** for one disk (i.e., either two-thirds of or equal to the number of revolutions per second). **Binit** then multiplies **maxpgio** by the number of disks doing paging. Unfortunately, **vminit** is called before **nswdev**, the number of disks with swap partitions, is set. So **vminit** always thinks that there is only one disk doing paging. The solution is probably to move the code that deals with **maxpgio** from **vminit** to **binit**.

the swap device

Conceptually there is only one swap device. If more than one disk has a swap partition, then the blocks of the swap device are interleaved among the swap partitions. Swap space for each text segment is allocated only once and shared by the processes running that program. Text is allocated in chunks of **dmtext** clusters. The swap areas for a process's stack and data segments are described by two separate maps. The first entry in a map points to a chunk of **dmmin** clusters. The second points to a chunk of **2*dmmin** clusters. The third entry points to a chunk of **4*dmmin** clusters. This pattern continues until the entries point to chunks of **dmmax** clusters, at which point the chunks don't get any bigger.

The paging and swapping code uses its own set of *swap buffers*, outside of the file I/O buffers described in Section 3.1. **Startup** allocates half as many swap buffers as there are file I/O buffers, with a maximum of 256. This fraction is not tunable in 4.2BSD. No pages are allocated to these buffers in **startup**. Instead the code that does I/O for paging and swapping links the buffer headers to the pages actually being paged or swapped.

4.2 Existing Tools

There are three tools in 4.2BSD that give information about the virtual memory subsystem: **vmstat**, **pstat**, and **ps**. **Vmstat** gives general information about the virtual memory system (plus a pile of other information, as we described in Section 3.2). **Pstat** displays information about the swap device and the text table. **Ps** optionally gives virtual memory statistics for each process that it displays.

vmstat

Vmstat gets most of its numbers from **vmmeter** and **vmtotal** structures kept in the kernel. When certain events occur (e.g., pageins), the kernel increments counters in **cnt**, which is a **vmmeter** structure. Once per second, the **vmmeter** function adds the values in **cnt** to **sum** (another **vmmeter** structure), clears **cnt**, and recalculates **rate** (a third **vmmeter** structure) using a smoothed averaging scheme. Every five seconds, **vmmeter** udpates swap counts and calls **vmtotal**, which updates **total** (a **vmtotal** structure) based on information in the text and process tables. We will not try to justify the rationale for this setup, other than to point out that some of the numbers (e.g., the amount of free physical memory, the number of runnable processes) are used by the kernel to make scheduling decisions. For this feedback, smoothed averages are better than raw counts because they are less likely to cause instability [Ferrari].

Many of the numbers that **vmstat** displays need some additional explanation. For example, "active" (as in "active virtual memory") means that the process is either runnable or has been sleeping for less than **maxslp** (typically 20) seconds. "Active virtual memory" is the sum of the virtual address sizes of all "active" processes. The number of "blocked for resources" processes is actually the number of processes that are sleeping with a "negative" priority (i.e., less than or equal to **PZERO**). The importance of negative priority is that the sleep can't be interrupted by signals. In fact, many of the processes that sleep at a negative priority are the same ones that are waiting for resources. However, some device drivers, for example, sleep at a negative priority because they are obeying a locking protocol. The label "blocked for resources" is therefore misleading.

The "number of processes in the run queue," the swap-related counts (which **vmstat** does not display), and the "active virtual memory" are strict counts. The "page" numbers and the "size of the free list," though, are averages. In the first line given by **vmstat** the "average" numbers are simple averages over the lifetime of the system (i.e., **sum** values divided by the time since the system booted). In the lines that follow, they are averages smoothed over the last 5 seconds (i.e., **rate** values). The "size of the free list" is an exception to this rule: it is always an average smoothed over the last 5 seconds. The "disk faults" numbers are another exception: they are the same transfer averages that **iostat** prints. The number of *attaches* is the average number of times that a process reclaimed a page that was owned by another process (we explained this in Section 4.1). The "reclaim" count that **vmstat** displays is another average, but it doesn't include attaches. The interrupts, system calls, and context switches counts are also averages. The interrupts count doesn't include clock interrupts or software interrupts. The "cpu" information is just like that given by **iostat**, except that **vmstat** doesn't tell how much time is spent executing *nice*'d processes.

Deficit is a kernel variable used by 4.2BSD to control swapins. When the swapper decides to swap a process in, it "reserves" memory for the process by increasing **deficit**. (It also increases **deficit** when in "desperation" mode.) **Vmstat** usually displays **deficit** as zero because **vmmeter** reduces **deficit**, simulating the effect of pageins, every time it is called.

One instrumentation structure that nobody displays is the **swptstat** structure. It keeps tabs on whether a process's page table shrinks, grows, or stays the same when it swaps. For example, 4.2BSD occasionally swaps out a process because there isn't enough room to grow the process's page table in core. This is called an *expansion swap*. Knowing that many processes' page tables grew when they were swapped out would be an incentive to increase **USRPTSIZE**, the number of pages allocated to user process page table entries (defined in **vmparam.h**).

other options to vmstat

As we explained in Section 3.2, **vmstat -s** simply dumps **sum**. The number of "fast reclaims" is the number of times **locore** (described in Section 3.2) can reclaim the page, rather than taking the extra overhead of going to C code to handle the page fault. The number of "reclaims from free list" is the number of reclaims made where the page had already been "freed." The count of "intransit blocking page faults" is the number of times a process faulted on a text page and slept while another process brought the page in. The "swap text pages found in free list" and "inode text pages found in free list" counts are the two components of the *attach* count that **vmstat** normally shows.

Vmstat -t produces numbers showing how much time is being spent doing reclaims and how much time is being spent handling all page faults. If **PGINPROF** ("pagein profiling") is turned on, **pagein** records the time-of-day when it is called and subtracts it from the time-of-day after it has satisfied the page fault. **Vmstat** digs out the two counters (total time spent doing reclaims and total time spend doing pageins) and displays the total and the average times.

Vmstat -f produces information about **forks** and **vforks**, which we'll explain in more detail in Section 5.1. Specifically, it gives the number of **forks** and **vforks** since the system booted, and it gives the total and average number of pages involved in each operation.

pstat -s, -x

Pstat -s produces information about the swap device abstraction. "Wasted" swap space is how much is allocated but unused by processes. The "avail:" line shows what chunks of swap space are available (i.e., it shows how fragmented the swap device is). **Pstat** produces this information by reading in the swap resource map (a **map** structure, as described in Section 2.3), the text and process tables, and the table of swap disks. It then mimics kernel code to calculate numbers like how much space is in use, how much is used by text segments, and how much is missing (i.e., unaccounted for).

The **-x** option causes **pstat** to display every entry in the text table. Not every field of an entry is shown. Only the following are printed: the entry's location, its flags (printed symbolically), the disk address of the segment's first swap block, the address of the first process using the text, the segment's *resident set* size (i.e., how much is in core), its total size, the virtual address of the inode that the text came from, the number of processes sharing the text, and the number of in-core processes sharing the text.

ps v

The **v** option to **ps** tells it to print virtual memory information for each process that it displays. We shall quickly note some quirks of this display. The "sleep time" and "residency time" are displayed as 99 if they are larger than 99 (they get up to 127 in the kernel). The **%MEM** field is the percent of memory described by **cmap**, not the percent of total system memory. **LIM** is the process's *maxrss* (*maximum resident set size*). If the process's maxrss is infinite (the default), **ps** prints it as "xx." Otherwise **ps** prints it in kilobytes. If the process is sharing its text segment with (n − 1) other processes, then the **RSS** field includes 1/n'th of the size of the text segment. (However, when **ps** prints the 4-character process state and decides whether the process is over its **maxrss**, it ignores text used by the process.) If the pager finds a page belonging to a process that has more than its maxrss in core, it frees that page without further thought. Also, if the system has less than **desfree** pages free and a process has more than its maxrss in core, the process is penalized by adjusting its priority.

4.3 Problems with the Existing Tools

In this section we'll run through a list of gripes about each of the tools, similar to what we've presented in previous sections. However, we shall also discuss one problem that is common to all of the instrumentation tools in 4.2BSD, which is how user programs and the kernel schedule sampling iterations.

vmstat

There are many numbers that we'd like to see from **vmstat** (perhaps as additional options). One of the problems with the current **vmstat** display is that the scan rate, paging rates, and number of free pages exist in a vacuum. There is no easy way to compare them with system parameters like **maxpgio**, **lotsfree**, and **desfree**. If nothing else, an option to **vmstat** should dump **maxpgio**, **minfree**, **desfree**, **lotsfree**, **fastscan**, and **slowscan**. Also, that same option might as well dump **dmmin**, **dmmax**, and **dmtext**.

Many of the numbers that **vmstat** produces are smoothed averages. As we mentioned previously, this is in part because some of those numbers are used in a feedback loop to control the system. Smoothed numbers are generally preferred in that case to avoid instability. Nevertheless, we'd also like to know how hard the system is being pushed--what some of the peaks are. So in addition to the average number of pages paged in and out, we'd like to know what the largest simultaneous number of pages in transit was during the last sampling interval. This would tell us how hard paging I/O is pushing the disks. We'd also like to know the smallest number of pages that were available during the last sampling interval. This would tell us how close the system really is to running out of memory.

Checkpage, **cleanup**, and **pagein** can keep track of the paging traffic. **Checkpage** initiates pageouts of dirty pages. It can bump the count of how many pageouts are being done at that time and keep track of the maximum count. **Cleanup** frees the pages written by **checkpage** and can decrement that count. **Pagein** would keep the count of pages being brought in and keep track of the maximum count. The pagein count is in a critical region because more than one process can be in **pagein** at once. However, only one process (the pager) is ever in **checkpage** or **cleanup**, so no mutual exclusion protection is needed for the pageout count.

The code that maintains **freemem** (the number of free pages in the system) is spread out through various parts of the kernel. The cleanest solution here is to use a macro to change **freemem**; this macro would keep track of the minimum **freemem**.

Another new number that we want is how many pages are locked in core by the kernel. For example, a small mbuf leak might not cause the system to crash, but it might lock up enough memory that performance would suffer noticeably. When we're first trying to figure out why a system is slow, knowing that many pages are pinned by the kernel would help point us in the right direction. Pages can be pinned in core either because they are being used by the kernel (so their type is **CSYS**), or because they are locked (e.g., for I/O). In the second case, the **c_lock** bit of the corresponding **cmap** entry will be set.

We'd also like **vmstat** to tell us the number of swapins, swapouts, and pages transferred in each case. Even though a process pages back in on demand after being swapped, the kernel brings in the process's page table and *u. area* (described in Section 5.1) at the time it swaps the process in. This information on swapping would give us a more complete picture of what the virtual memory subsystem is doing. Finally, as we mentioned in Section 4.2, it'd be useful to know the number of expansion swaps the system has done, so that we can tune **USRPTSIZE**.

There are a couple of numbers that **vmstat** prints now whose usefulness is questionable. As the size of physical memory in Berkeley UNIX systems grows, it takes longer and longer for the imaginary clock hands of the pager to make one revolution through memory. Thus there is little point in keeping track of how many revolutions the hands have made. Also, we question the usefulness of "active virtual memory." Because a process is not likely to access all of its virtual memory at once, this number only approximates the load on physical memory (i.e., the amount of core being used). Of course, before we dump the notion of active virtual memory, we should check how accurate it really is. We can do this by adding the resident set sizes for all the processes in the system and comparing the sum with the "active virtual memory" number.

As we shall discuss shortly, sampling the system at the right time is an important issue in instrumentation. In 4.2BSD the *callout* code provides an alarm-clock mechanism in the kernel. You call **timeout** with a pointer to the function you want called when the alarm goes off, and you say how many clock ticks later you want the alarm to ring. When the alarm goes off, **softclock** calls your function, passing two parameters. First is a single argument that you'd supplied to **timeout**. The second parameter tells how many clock ticks ago the function *should* have been called. In the case of virtual memory metering, **schedcpu** is called every second via this mechanism, and **schedcpu** calls **vmmeter**, which does the metering. Unfortunately, when **schedcpu** resets its alarm to go off one second later, it ignores the number of ticks that it was late in being called. Thus if for some reason **softclock** is late calling **schedcpu**, the metering code falls behind and never catches up again. How serious this problem is depends on how busy the system would have to be before **schedcpu** got called late.

A more important problem is how **vmmeter** decides when to call **vmtotal**. Rather than calling **vmtotal** every fifth time, **vmmeter** looks at the time of day. If the number of seconds in the time-of-day value is a multiple of 5, then it calls **vmtotal**. Thus if **vmmeter** is called when the time-of-day is almost a multiple of 5 seconds, and then **vmmeter** is called when the time-of-day is just barely past a multiple of 5 seconds, **vmtotal** gets lost in the dust. This causes the instrumentation numbers--one of which is used for controlling system load--to stick for an extra five seconds, and it means that when **vmtotal** is finally called, the numbers for swapins and swapouts will be twice what they should be.

We also dislike the way **vmmeter** decrements **deficit**. **Vmmeter** assumes that, on the average, one half of a maximum-size kluster is brought in (klusters are described in Section 4.1) and that one-half **maxpgio** pageins are done (i.e., half for pageins and half for pageouts). If this number of pages is less than 10 percent of deficit, though, **vmmeter** reduces deficit by 10 percent. We have yet to find a convincing explanation for this strategy, and we think that "black magic" should have little place in kernel code.

A consistency quibble is that the values for swapins and swapouts in **rate** are simple counts of those events over the last five seconds. Every other value in **rate** is an average smoothed over the last five seconds. If a program displays those counts, it should probably divide through by 5. Otherwise, the user would have to remember which of the numbers displayed are averages and which are total counts.

An interesting property of the instrumentation code involves swapped out shared text segments. If all processes sharing a program (e.g., the **emacs** editor) are swapped out, then the corresponding text segment is swapped out as well. If some new process want to run that program, 4.2BSD loads the text segment from the swap device, rather than from the file system. Thus, a process can cause the system to claim that it swapped pages in, even though the process itself was never swapped.

other options to vmstat

The only problem with the **-s** option is that it includes numbers that should go somewhere else. For example, there is little point in displaying the number of pseudo-DMA interrupts in **vmstat**, and there is even less point in displaying statistics about the **namei** cache there. Those numbers should go in **iostat**.

Vmstat -t has problems that are much worse. The first problem is that it uses numbers that are not always calculated by the kernel. Unless the kernel was compiled with **PGINPROF** defined, the variables used by **vmstat -t** exist but are not maintained. This means that **vmstat -t** will always produce results, but you have no way of knowing whether the numbers mean anything. Instead, the variables should only be defined if **PGINPROF** is defined, and **vmstat** should assume that if it can't find the variables, then the information isn't available. Also, the kernel routine that computes the pagein times is broken. It currently assumes a 60 Hz system clock and that **lbolt** contains the number of clock ticks between exact seconds. To work properly, this code need only use **time.tv_usec**, which is the microsecond portion of the system time variable.

pstat -s, -x

Many of the bugs in **pstat -s** result from the mimicry of kernel code inside pstat. One problem with this mimicry is that it's liable to mistakes. For example, the kernel computes in pieces the swap space needed by a process. **Pstat** tries to bring those pieces all together, but in a couple of places it forgets about rounding and units conversion. What's worse is that **pstat** won't easily track changes to the kernel. Another problem is that **pstat -s** has to read 4 different data structures to do its job. This increases the likelihood that the information will change while **pstat** is reading it, leading to incorrect results. The only purpose of this full-scale mimicry is to find "missing" swap space. If, however, most of the time the space is "missing" because of bugs in **pstat**, then you might as well go to a simpler scheme in which kernel code (in **vsexpand**, **vsxalloc**, **vgetswu**, **vsxfree**, **vrelswu**, and **swfree**) keeps counts of how much space is free, how much is in use, how much is used for text segments, and how much is wasted. To get fragmentation information **pstat** would still have to read in the swap resource map and try allocating chunks of different sizes, though this at least reduces the number of reads from kernel memory. Also, under this scheme the kernel and **pstat** can share common code (**rmalloc**, **rmfree**), which means that **pstat** need only recompile to track kernel changes.

Our only complaint with **pstat -x** is that it, like other **pstat** options, displays flags symbolically. We discussed this issue in Section 3.3.

ps v

One problem with the **v** option is its assumption that the process's default maxrss is infinite. This was the default in 4.2BSD, but in 4.3BSD the default will be based on the system's physical memory size. If we want to distinguish processes that have a default maxrss from those that do not, then **ps** must change to recognize the new default. Because "infinite" resource limits aren't really infinite in Berkeley UNIX, just very large, there is no point in distinguishing processes with an "infinite" maxrss.

sampling

Both the kernel and iterating user programs (**vmstat**, **iostat**, and **netstat**) use a scheme in which an alarm goes off, they gather information, and then they reset the alarm. The kernel does this in **schedcpu** by calling **timeout** when it has finished. A user program (we'll use **vmstat** as an example) does this by calling **sleep** when it has finished one iteration. This means that the system is sampled every 1-and-some-fraction seconds, rather than every second. On the one hand, this causes an obvious inaccuracy in 4.2BSD instrumentation, especially when the system load is high. In the long run it biases the results towards the lightly loaded case because the system misses data points from the heavily loaded case. On the other hand, if the system *is* heavily loaded it will spend more time getting "useful" work done instead of doing system measurements. We expect that at the kernel level the inaccuracy may be only a clock tick or two anyway (though this should be checked). At the user level, though, the resulting inaccuracy is simply intolerable. We can illustrate this problem even on a lightly loaded system: we tell **vmstat** to iterate every second, and we watch numbers that should change every 5 iterations change five to ten percent of the time after only 4 iterations.

So we need additional measurements ("meta-instrumentation") before deciding whether **schedcpu** needs fixing. The fix would be to move the call to **timeout** to the front of **schedcpu**, rather than the end. The user-level programs need to replace the call to **sleep** with a call to **sigpause**. A call to **alarm** should go at the top of the sampling loop, plus there must be code to handle the case where the alarm goes off before the process has reached **sigpause**.

In user-level code, the **alarm** signal handler should set a flag saying that the alarm went off; this flag should be cleared just before calling **alarm**. The code immediately preceding **sigpause** should block **alarm** signals with **sigblock** and test that flag. Only call **sigpause** if the flag is not set. Otherwise, just unblock **alarm** signals and go back to the top of the loop.

miscellanea

We would like to see one change to virtual memory-related logging. In 4.2BSD a process can get killed because there is no swap space. We would like log messages to distinguish the case where there was simply not enough space from the case where there was enough space but it was too fragmented.

We'd also like to know exactly how much physical memory is available to user processes (i.e., the size of physical memory less memory taken up by kernel text, kernel tables, and I/O buffers). This number would be useful to people running large programs who'd like to avoid paging.

Our final comment has to do with the calculation of **rate**, the calculation of the load average (to be discussed in Section 5.1), and how **vmmeter** reduces **deficit** (discussed in Section 4.3). In the first two cases, the 4.2BSD kernel is computing a smoothed average. The load average computation, which uses exponential coefficients, was specifically designed so that the system would "forget" a specific fraction of the system load after a specific number of seconds. The computation of **rate**, though, uses linear coefficients (1/5 of **cnt** plus 4/5 of the previous **rate**), which seems to put it in the category of "wild-guess heuristic." Similarly, the ten percent reduction of **deficit** in **vmmeter** seems poorly motivated. It would be nice if these two parts of the kernel could be better justified.

5. Processes and the CPU

5.1 How it Works

First we shall explain how UNIX commands are executed and the states that a process goes through during its lifetime. We'll then explain how 4.2BSD schedules processes, and we'll explain the basic UNIX protection mechanisms.

process states; fork and exec

Two data structures describe a process in 4.2BSD. The first structure is the **proc** (process) table. It contains all the information that the kernel needs even when the process is swapped out. The other structure is the process's *u. area* ("user area," pronounced "you-dot area"). This structure is swapped with the process and contains information that is only needed when the process is swapped in. The process table entry has the swap address of the u. area when the process is swapped out. It has a pointer to the first page table entry for the u. area when the process is swapped in. The u. area has a pointer to the process table entry.

In UNIX new programs are executed via **fork** and **exec**. When a process **forks**, UNIX duplicates that process, giving the *child* process a copy of the *parent* process's data and stack segments, open file descriptors, and resource limits. The child and parent share the text segment. The parent and child processes know which they are by looking at the return value of the **fork** system call. The child process then **execs** the file that holds the new program. A variation of this scheme is for the parent to use **vfork** instead of **fork**. In this case the kernel doesn't copy the parent's data and stack segments for the child. Instead, the child is given the parent's pages. The child is expected to almost immediately **exec** a new program or die trying (i.e., **exit**). When either of these two events happens, the parent is given its pages back.

The states that a process may be in are shown in Figure 5.1. A process's first state is **SIDL**. This state is used when the kernel is filling in the process table entry. It reserves the entry but shows that the process isn't runnable yet. Once the process has all of its resources (e.g., page table entries, swap space) and the process table entry is all filled in, the process's state changes to **SRUN**. The process can go from this state to one of three other states. If the process stops, either for tracing or because of a signal, then the process enters the **SSTOP** state. If the process enters the kernel and has to wait (e.g., for I/O or for a signal), then the process is put in the **SSLEEP** state while it is waiting. When the process **exits**, its resources are taken away from it and its state changes to **SZOMB** ("zombie"). The process stays a zombie while waiting for pending I/O to complete and while waiting for the parent process to *reap* it. Reaping is the way that a parent process finds out what resources (e.g., how many CPU cycles) its children used, and it provides a way for a child process to return a status code to its parent.

scheduling

In 4.2BSD the priority of a runnable process is based on how much CPU time it has used lately, the system load, and the process's *nice* value. Numerically lower priorities are better than higher priorities. As a process runs, the kernel recalculates its priority every four clock ticks (each tick being 10 milliseconds). The priority increases linearly as the process runs, and it decreases exponentially while the process isn't running. The greater the system load, the slower the priority will decay. Every 100 milliseconds **roundrobin** forces the system to look for the process with the best (i.e., lowest) priority. Once per second **schedcpu** recomputes the priorities of all processes that are runnable or have been sleeping for less than one second. The *nice* value, which ranges from −20 to +20, moves the priority up or down independent of the process's CPU usage.

In 4.2BSD **loadav** computes a system *load average* every five seconds, similar to what was done in Multics [Saltzer & Gintell]. Actually, the kernel computes three exponentially smoothed averages of

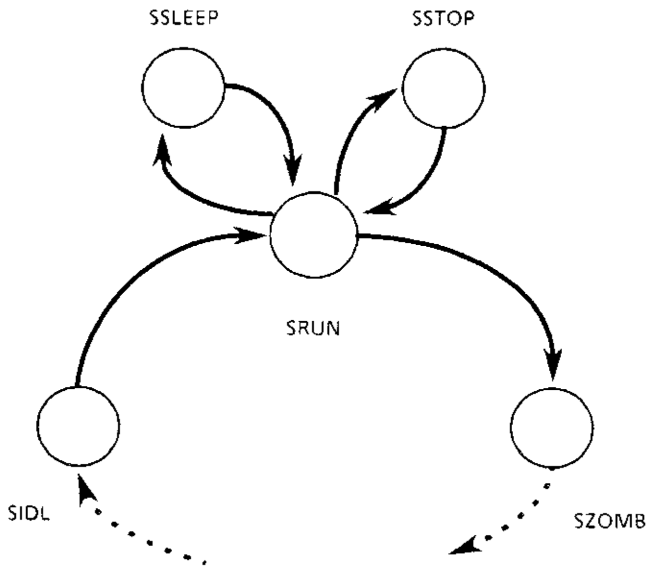

Fig. 5.1: the life cycle of a process

the number of runnable jobs. The first is the average over the preceding minute, the second is the average over the preceding five minutes, and the third is the average over the preceding fifteen minutes. The 1-minute average is used to control how quickly a process regains priority when not running.

security

Each Berkeley UNIX user is associated with a user ID and one or more group IDs. Each file, on the other hand, is associated with exactly one user ID and exactly one group ID. Each file has three sets of three permission bits (one bit for read, one for write, and one for execution). When a process tries to access a file, there are three possible cases. If the process and file are owned by the same user, then the kernel uses the first set of permission bits. If the user IDs are different but the file's group is among the groups for the user, then it uses the second set of bits. Otherwise, it uses the third set. The exception to this rule is: if the process is owned by **root** (the *super-user*), then the process always passes the security check.

Actually, each process has two user IDs associated with it: an *effective user ID* and a *real user ID*. Normally these two values are the same. However, if a process **exec**s a file that has the **setuid** ("set-you-I-D") bit set, then the process's effective user ID is changed to the user ID of the file's owner. For example, the **rcp** program runs "setuid to root" so that it can communicate through a reserved Internet port. Similarly, a process has an *effective group ID* and a *real group ID* (in addition to belonging to a list of groups). The effective group ID can change by **exec**'ing a file with the **setgid** bit set. This mechanism allows "trusted" programs to access an important resource (e.g., a database), while keeping untrusted programs out. In fact, an example of this use will come up in Section 6.4.

5.2 Existing Tools

The 4.2BSD distribution provides a variety of process-related tools. **Ps**, **who**, and **rwhod** do some processing of system information, whereas **pstat** just pretty-prints system tables.

ps

Ps's purpose in life is to provide interesting numbers based on the contents of the process table and u. areas. Some of its options (**a**, **g**, **t**, and **x**) control which processes **ps** reports on. Other options (**c**, **e**, **l**, **s**, **u**, **v**, and **w**) control what information is printed about the process.

Ps tries to print the name of the program that the process is running, along with its arguments. It may also print the program name that will be used for system accounting, which may differ from the other name. The swapper and pager processes don't have associated command names, but they have well-known process ID's, so **ps** can still print a reasonable "name" in those two cases. When a process is exiting, though, it is no longer associated with a program. In this case **ps** prints the name "<exiting>." If the process has made it all the way to zombie status, the name is "<defunct>."

The "state" letter that **ps** prints is determined by the process's state, its flags, and its priority. If the state is **SSTOP**, then **ps** prints "T." If the state is **SSLEEP**, then **ps** looks at the priority. If the priority is "positive," then **ps** prints either "S" (if the process is active) or "I" (if it is not). (We discussed "negative"--non-interruptible--priorities and "active" processes in Section 4.2.) If the process is in state **SSLEEP** and its priority is "negative," then **ps** looks at the **SPAGE** flag for the process. If that flag is set, then **ps** prints "P." Otherwise, it prints "D." If the process's state is **SRUN**, then **ps** prints "R." Finally, if the process's state is **SZOMB**, it prints "Z." This means that the processes that **vmstat** lists as "blocked for resources" are the same ones for which **ps** prints a "D" state, except that the **vmtotal** function (which maintains those numbers for **vmstat**) ignores the pager and the swapper. This also means that **ps**'s documentation is misleading for the same reason that **vmstat**'s is: the process isn't necessarily waiting for the disk or for any other resource, it's just waiting at a numerically low priority.

If the process being displayed is swapped in, the **ADDR** field (**ps l**) is the number of the page frame that starts the process's u. area. Otherwise **ADDR** is 0. The **%CPU** field is a smoothed average telling what percentage of the CPU the process has used recently. The kernel keeps this number solely for **ps**'s use. The **CP** field, however, is used by the kernel to compute process priorities. It is incremented with every clock tick that the process is running, and it decreases according to an exponential smoothing function based on the 1-minute load average.

pstat

Given a page frame number, **pstat -u** dumps selected fields from the u. area stored there. You can get the page frame number from either **ps l** or **pstat -p** (the **ADDR** field in either case).

The **-p** option to **pstat** causes it to dump certain fields for each entry in the process table. None of this information is symbolic. Even the state is given as a decimal integer. We shall briefly mention some trivia about the fields of **pstat -p**. Only the least significant 16 bits of the process's flags are given; the rest of the flags field is dropped. The "signals received" field refers to signals received but not yet processed by the process. **SLP** and **TIM** (time sleeping and time in core) are often 127, which is the limit the 4.2BSD kernel places on them. The **ADDR** field displays the address of the process's u. area. If the process is swapped out, the address is a block number on the swap device. Otherwise it is the number of the page frame at which the u. area starts. You can tell whether the process is swapped in or out by looking at the flags. If the **SRSS** ("resident swap size at last swap") is zero, that doesn't necessarily mean that the process has never been swapped out. It's possible that the pager had taken all of the process's pages before it got swapped out. The **LINK** field is used for more than just connecting runnable processes. For example, it is also used to connect all of the processes that are waiting on a given event.

When a process calls **sleep**, it gives an integer specifying what "event" it is waiting for. This integer is typically the address of a data structure associated with the function being performed. For example, when the pager is waiting for something to do, it calls **sleep** with the address of its own entry in the process table. The code in **sleep** puts the process in a chain of processes that are waiting for the event. When a process does a **wakeup** call on that event, the kernel puts all of the processes that were sleeping on it back into the run queue.

If you use **pstat -pa**, you get all of the entries in the process table. Otherwise you just get the "active" entries, which in this case means all the entries that are in use.

who

The "who" command lists the users on the system. It does this by scanning **/etc/utmp**, which lists for each terminal who is logged on, when that person logged on, and, if the terminal is associated with a pseudo-tty, what machine the user is logged on from. Another program, **rwhod**, broadcasts similar information on each network to which the system is connected. The difference is that **rwhod** broadcasts each user's *idle time*, rather than the remote machine the user is logged onto. It computes the idle time by **stat**'ing the terminal that the user is logged on and checking its last modification time (recall that there is a file for each UNIX device). In the same packet **rwhod** also broadcasts the machine's load average and the time it booted at. Two programs in 4.2BSD read the information that neighboring machines have broadcasted. **Rwho** prints the **who**-like information (user, tty, idle time). **Ruptime** prints the machine's name, how long it's been up, the number of users on it, and its load averages. Both **rwho** and **ruptime** normally consider only "active" users, which are users with idle times of less than one hour.

A **rwhod** process spends most of its time listening for packets from other **rwhods**. When it gets a packet from the **rwhod** on, say, ucbernie, it copies that file into **/usr/spool/rwho/whod.ucbernie** for use by **rwho** and **ruptime**. Every so many minutes an alarm goes off and it broadcasts a packet of its own.

w

Using **/etc/utmp** and the process table, the **w** program tries to tell what each user on the system is doing. Its first line gives the system load average, the number of logged-on users, and how long the system has been up. (If **w** is invoked as **uptime**, that's all it prints.) For each logged-on user, **w** then prints the user's login ID, what terminal the user is on, when the user logged on, the user's idle time, an estimate of the CPU time used by the user's "job" (login session), an estimate of the CPU time used by user's currently running processes, and the name of the "current" program. **W** associates processes with jobs by looking at the process's *controlling tty*. The per-job time is the sum of the time used by the active processes and the active processes' children. **W** picks the "current" program based on process ID's and whether the program is receiving interrupts from the terminal.

the accounting package

We consider the 4.2BSD accounting package to be a facet of overall system instrumentation. We back up this claim by noting that if commonly used programs are slow, then the system as a whole will be slow; we can use the accounting package to instrument the system by finding out which programs are used the most and what system resources they consume.

When a process **exec**'s a file, the kernel records the name of the program or command file ("shell script" in UNIX jargon) being run. As the process runs, the kernel records in a structure what the process requests from the system. Example fields from this **rusage** ("resource usage") structure are the CPU time spent in kernel mode (*system time*), the CPU time spent in user mode, the number of disk block transfers, the number of messages received, the number of reclaims (defined in Section 4.1), the number of pages faulted in from disk, and the number of "voluntary" context switches. Some of the accounting information is recorded at the same time system-wide values are recorded (e.g., number of pages faulted in from disk). Other accounting information is recorded independently from system-wide instrumentation (e.g., number of disk block transfers). (If there is little room in **/usr/adm**, the kernel instead announces that it has suspended accounting. It then drops accounting records on the floor until space frees up again.) The **sa** program produces accounting reports and condenses **/usr/adm/acct** still further.

5.3 Problems with the Existing Tools

ps

Ps, so far as it goes, does pretty much what we want it to. However, 4.2BSD does not have a program that gives a summary of all existing processes. Thus we would like a separate tool that displays the number of runnable or nearly-runnable processes, the number of processes that have been sleeping or stopped for a moderate time, the number of processes that have been sleeping or stopped for a long time, and the number of zombie processes. The number of zombie processes might point to an error in a server that forgot to reap its children. The number of sleeping processes might point to idle server processes that should be combined into one program, as was done at Berkeley after 4.2BSD was released [Leffler *et al* 84]. The number of long-term sleepers (or stopped programs) might point to users who don't log out when they leave. For system administration, we would like to see how many processes are running with "negative nice" (i.e., artificially high priority). Most of this information is available with the numbers that the 4.2BSD kernel already keeps. The only change would be to differentiate between short-term sleepers, medium-term sleepers, and long-term sleepers (treating a stopped process the same as a sleeping one). The kernel would have to record the process's "sleep time" at least until it hits the "long-term" category.

The numbers that distinguish medium-term from short-term and from long-term need to be thought about more and perhaps experimented with. The boundary between (nearly) runnable and short-term sleeper might be a few seconds. The boundary between short-term sleeper and medium-term sleeper might be on the order of minutes. The boundary between medium-term sleeper and long-term sleeper should probably be at least an hour.

Of our suggestions for changes to **ps**, most correct minor bugs. One trivial bug is that **ps** thinks that sleeping at priority **PZERO** is interruptible. Well, **ps** got the boundary condition wrong. As we explained in Section 4.2, a sleep at **PZERO** is not interruptible. Another small bug involves the program name that **ps** prints. The 4.2BSD kernel does not clear the **SWEXIT** flag when the process becomes a zombie. **Ps** checks this flag before it checks the process's state, which means that **ps** prints "<exiting>" even if the process is a zombie. Another problem with program names is that the area **ps** looks at is writable by the process. Thus we occasionally see "commands" that were given as

```
rcp /usr/bin/ps ucbernie:/usr/bin
```

displayed by **ps** as

```
rcp /usr/bin/ps ucbernie /usr/bin
```

because of processing by **rcp**. Of course, the process could have put anything it wanted there. One way around this administrative loophole is the **-c** option, which tells **ps** to print the command name used for accounting (which is not changeable by the process).

A more general problem with **ps**'s implementation is that it doesn't read the process table in all at once. Instead it reads in a piece, decides what entries to keep, stashes them away, and reads in another piece. As we shall discuss in Section 6.4, a typical problem with 4.2BSD instrumentation is that the numbers being displayed can change from underneath the program displaying them, leading to inconsistencies or other weird results. Reading the process table in pieces makes **ps** more susceptible to this problem than if it read the entire table at once. We could argue that the memory requirements for reading in the entire table are excessive. On the other hand, we note that **w** and **pstat** read in the entire process table, so we don't think that this argument holds up.

If we have a tool that tells how many processes are in what state, then we'd also like **ps** to identify processes that are long-term sleepers (or are stopped for a long time). This would let us get a better handle on what processes are tying up space in the process table. **Ps** would not necessarily have to flag such processes explicitly. If the kernel only records the sleep time up until the process reaches the "long-term" category, then we can use **grep** to find the long-term processes via a string search.

Finally, **ps**'s performance is bad. As we shall explain later, **ps** is the tool of choice for finding what processes are bogging down the system. Unfortunately, we usually have to ask ourselves whether we want to run **ps** and wait a long time, or whether we'll just grit our teeth and hope the load goes away soon. If we look at a profile of **ps uax**, we find that the CPU time used by **ps** is pretty much proportional to the number of entries that **ps** records information about. Thus a useful addition to **ps** would be an option telling **ps** to ignore swapped-out processes: they essentially contribute nothing to system load, and they seem to be about one-third of the processes on the machines at Berkeley.

Ps sifts through the process table, calling **save** for each entry that it wants to keep. After that, it sorts the entries and displays them. Thus ignoring swapped-out processes would just be another test to skip calling **save**.

pstat

Once again we have many difficulties with **pstat** to report. The single bug that we found in **pstat -p** is that it was not printing the correct value for the **ADDR** field. This happened because **pstat** failed to track a change in the kernel. Also, we would like the **-p** option to print all of the flags in the process table entry, not just the least significant 16 bits of the flags.

There were three small problems with **pstat -u** and one very big one. The first small problem was that **pstat** was trying to read the u. area from **/dev/kmem** (kernel virtual memory) instead of **/dev/mem** (physical memory). Of course, a process's u. area is accessible from kernel virtual memory, but the page frame number given to **pstat -u** only makes sense when dealing with physical memory. The second problem was that the array for system call arguments had grown and **pstat** wasn't printing the last 3 arguments. The third small problem was that **pstat** was printing the **u_ssave** structure twice and not even labeling the first copy. Perhaps this was a hack to get around the big problem, which is that the u. area no longer fits inside a single cluster. This is a problem first of all because the u. area is not necessarily stored in sequential page frames. So knowing the frame number of the first page of the u. area doesn't help **pstat** find the next cluster of the u. area. Second, simply changing **pstat -u** to use, say, a process ID instead of a page frame number won't work either. One use for **pstat** that we haven't mentioned is to examine the u. area in user program dumps, for which you give zero as the frame number. Perhaps the solution is for **pstat** to accept a process ID instead of a frame number, but to read from the start of the core file if the user doesn't give a process ID. (The 4.2BSD **pstat** complains about not getting enough arguments if the user doesn't give a process ID.)

who

Rwho and **ruptime** decide that a system is down (and that the **rwhod** file for it can't be trusted) if the **rwhod** file has a timestamp that is too old. Of course, "too old" is relative, depending on how often **rwhod** broadcasts. In the last 18 months, the broadcast interval at Berkeley has changed twice, from 1 minute to 5 minutes to 3 minutes. Each time both **ruptime** and **rwho** had to be edited to track the change. Instead, **rwhod.h**, the header file shared by **rwhod**, **rwho**, and **ruptime**, should define the broadcast interval, a macro for determining a system's status (based on the timestamp age and the broadcast interval), and the maximum idle time for an "active" user.

Another problem with **rwhod**, **rwho**, and **ruptime** comes from **rwhod**'s broadcasting only the first 1024 bytes worth of user names. This is a reasonable tactic to avoid wasting network bandwidth when one or more systems are presumably busy. Unfortunately, **rwhod** does not flag the cases where there were more users than would fit in one kilobyte. The **rwhod** packet should include such a flag. **Rwho** and **ruptime** should note those systems for which the flag is set.

Our final comment is that **who** should (perhaps optionally) display the idle time for each user. This time would be more up-to-date than using **rwho**, the list of users would not get truncated (which is possible using **rwho**), and it would take this function away from **w**, which, as we shall explain next, we would like to eliminate.

w

The main use of the **w** command is to find out who is doing what. (It might also be used to list user idle times, but that is something we'd rather have done by **who**, as we've just noted). The problem is that **w** is simply not accurate enough. It bills processes according to the controlling tty, even if the user who started the process is not the one currently on that terminal. And even if the user is the same, **w** still doesn't always get the right command. Also, **w** only looks for a single command. If someone is running one or more background processes, **w** won't find them, though they may be contributing the most to system load. We could add yet another option to **ps**, this one causing it to sort the output by user. The problem with that scheme is that the results won't be any more useful than the output from **ps u**. The solution seems to be to junk **w** and use **ps u**, using the previously discussed option not to display swapped-out processes.

the accounting package

Before we discuss the problems with the current accounting package, we should point out that we have not looked at the entire package. In particular, we have not tried to find bugs in **sa** or in the mechanism that writes accounting records to **/usr/adm/acct**. We have, however, considered what the accounting records contain and how the kernel gathers resource information. First we shall consider numbers that the system gathers incorrectly. We'll then discuss additions to the records.

One flag in an accounting record tells whether the process used super-user privileges (e.g., to allocate a disk block when there is less than **minfree** percent left, as described in Section 3.1). This flag is set in the kernel by the **suser** routine, which checks the process's effective user ID. Unfortunately, there is a long list of routines that check the process's user ID themselves, instead of going through **suser**. One reason for avoiding **suser** is that it sets the process's global error flag if it fails the super-user test. This may be an unwanted side effect. Another reason not to use **suser** is when the check is against the real user ID, instead of the effective one. What we need are two routines. One checks the effective user ID, the other checks the real user ID. Each one should take an argument that tells whether it should set the process's error flag if the super-user check fails. However, changing the kernel to use those routines is not a simple mechanical string replacement. Calls to those routines should only be made if the process has failed all other permission tests. Thus code like:

```
    if (u.u_uid != 0 && u.u_uid != ndp->ni_pdir->uid)
```

should look more like:

```
    if (u.u_uid != ndp->ni_pdir->uid && !suser(SET_ERROR))
```

Another problem is that 4.2BSD bills a process's resident set size (the **p_rss** field in the process table entry) entirely to the data segment, rather than billing the data segment size to the data segment and the stack segment size to the stack segment. Fixing this inaccuracy requires that the kernel keep track of the in-core data and stack sizes separately. This should require little work. The routines that maintain **p_rss** already distinguish the text segments from the data and stack segments. Separating data from stack would only be another **if** test.

An inaccuracy that is harder to fix is how 4.2BSD bills disk writes. When a process **write**s some bytes, the block-device buffer usually sits in core for a while before UNIX writes it out to disk. This strategy can save on disk transfers. For example, consider a **write** that covers less than a complete block. If it is followed by another **write** to the same block, then UNIX can do two **write**s with only one disk access. I/O accounting is done at the disk access level rather than at the system call level. This is because the disk access is significantly more expensive for the system than the system call is. The problem is that if process A writes to a disk block and then process B writes to the same block soon afterwards, process A will get charged for the entire disk access, but process B will not get charged anything. It is not clear what to do about this inaccuracy. The only "fair" scheme is to bill both processes, but then the system would bill for more disk transfers than it actually did. Note that if the system bills the process when the buffer is actually written to disk, it could find itself trying to bill a process that has already exited. Thus the existing scheme seems to be the only workable one. In any event, though, it seems rare for more than one process to be writing to the same disk block at nearly the same time, so we believe that the problem is not worth pursuing further.

Another hard problem is how to deal with **exec**s. When a process does an **exec**, its resource information is not cleared. Unfortunately, only the last file to be **exec**'d is associated with the accounting record for that process. One solution would be to generate an accounting record when the process does an **exec**. However, if we assume that most processes only **exec** one program and that they do the **exec** almost immediately after being **fork**ed, then the system would generate twice as many accounting records and gain little additional information. So once again, the problem seems small enough to ignore.

Another naming problem is that the name in an accounting record is only the final name of the file's path. For example, both **/bin/csh** and **/usr/new/csh** are recorded as **csh**. The **exec** code could record the entire path for the program, but we'd like to avoid allocating dynamically sized buffers, and a buffer large enough to hold most path names would waste kernel memory. The solution that we propose is to record the "reverse path" of the program. For example, the two previous cases would be recorded as **csh/bin/** and **csh/new/usr/**. This scheme provides more information than just **csh**, and it should differentiate many cases even when the entire path doesn't fit in the accounting record. For example, assuming a limit of ten characters, we could reasonably conclude that **csh/new/us** is **/usr/new/csh**. We don't recommend that the kernel try to figure out the full path name if it wasn't given one. In the case of the standard UNIX commands, the kernel will usually get a full path name to work with. The other cases (not standard commands) are not as important to us.

The tail end of **execve** copies the program name from the **namei** argument structure into **u.u_comm**, which holds it until the process exits. The catch is that if it's a shell script being **exec**ed, the name--but not the complete path--of the shell script is stashed away and then later copied into the **namei** argument structure. Thus the change to store the reverse path must take care of both cases: regular program and shell script.

There are two new sets of numbers that we'd like to see in the accounting records. One set, the number of voluntary and involuntary context switches, is already kept by the kernel, but it isn't included in the accounting record when the process exits. These numbers would be useful on servers. If a machine is mostly used for non-interactive work, then it might be useful to lengthen the system time-slice (i.e., the time between calls to **roundrobin**). This lengthening would only really help, though, if most of the processes' context switches are involuntary, rather than voluntary. Hence we'd like to see these numbers in the accounting package.

The other set that we want is the number of messages sent and received by the process, as well as the number of bytes sent and received. The routines **sosend** and **soreceive** already record the number of messages sent and received, though these counts are not in the accounting record. To get a complete picture of how much IPC load the process caused, however, we need to know both the number of messages and the number of bytes.

miscellanea

As with previous sections, we want the system to log cases where it ran out of some important resource. The code in 4.2BSD already does a good job of this. Our only request is that **fork1** (in the kernel) log attempts by a user to create more than **MAXUPRC** processes. This failure probably occurs infrequently (in contrast with the problem of opening too many files at once) and would be useful for tuning **MAXUPRC** on systems with special work loads.

Also, we should point out another general problem with instrumentation in 4.2BSD. Information about, say, the states of the CPU and of disks is obtained by sampling every so many milliseconds, even though the systems at Berkeley have a hardware clock with microsecond resolution. The instrumentation code assumes that whatever the state of the system when it is sampled, that's the state it was in for the entire clock period. If the system state changes often with respect to the sampling rate, then the samples we get are not going to be accurate. We have not dealt with this issue for this project, though it is certainly an important topic for further research.

Another timing problem is that if the sampling rate is not relatively prime to the rates of the periodic system functions that influence the variables being measured, then the samples will be biased. Fortunately, 4.2BSD supports an "alternate profiling clock" that at Berkeley runs at 109 Hz (the normal system clock runs at 100 Hz). The kernel uses this clock to keep track of how often the CPU is in what state and how often the disks are busy. We therefore consider this problem to be solved.

6. Comments on Implementation

In addition to the implementation details that we've given so far, we have some general comments about implementing the ideas in this paper. First we shall comment on the UNIX "religion" and on what is appropriate in a UNIX system. Then we'll comment on current research at Berkeley that will affect instrumentation of future systems. Our third topic is that of test suites to validate the instrumentation code. Fourth, we shall recommend ways to get numbers from the kernel to the user level. We'll then discuss the construction of user instrumentation programs and general software engineering tips. Finally, we shall comment on documentation.

6.1 To Hack or Not To Hack

The ideas presented in this paper are only a portion of those considered while working on this project. We have tried to home in on the utilization and administration of critical system resources, such as disks and memory. Until someone tries out the ideas in this paper, though, we can not be sure which ones are worth keeping. People should not add code to UNIX just because they think it will be useful in certain cases. There are three reasons for this, based on the accepted notion of UNIX as a small, elegant operating system. The first and least important reason is aesthetics. There's no point in altering good clean code unless you get some return for your efforts. The second reason is performance. Instrumentation is system overhead. Although we have presented mostly cheap counting schemes, there is the possibility that the proposed changes will noticeably degrade performance. The final and most important reason is to reduce complexity. More code means more chances for bugs, and it often means more work when making future modifications. Instrumentation code is useless, even harmful, if it gives wrong numbers. Berkeley UNIX has already been criticized for its size and complexity. Changes should be implemented only if they are worth their debugging and maintenance costs.

We therefore suggest that the people who implement these ideas carefully consider which ideas they choose to work on. We suggest that the following projects be given higher priority than the others:

- fix bugs in existing code
- add gateway information to the IP statistics
- instrument the different types of network traffic (e.g., mail, files, **rwho** packets)
- record traffic between different hosts (for network configuration)
- add an option to **ps** to print socket PCB addresses
- instrument all disk drives, not just the first four
- add per-partition numbers to **iostat**
- display disk utilization in addition to the number of transfers per second
- use **alarm** and **sigpause** in user programs, instead of **sleep**
- replace **w** with **ps u**, and add an option to **ps** to ignore swapped-out processes
- fix the accounting package to record message load

Finally, we suggest that the resulting system be profiled to make sure it's not spending too much time monitoring performance.

6.2 Changes to Berkeley UNIX

Berkeley UNIX is a research effort, which implies that it frequently provides new services. We have briefly considered some proposed enhancements to Berkeley UNIX, and we have a couple of suggestions for instrumenting them.

A substantial research project at Berkeley has been the development of a remote file system [Hunter]. This project is based on the notion of mounting a remote machine's disk partition on a local directory, in much the same way UNIX already mounts local disk partitions on local directories. There will be two different numbers with which we would like to measure such a system. The first is the percentage of network traffic that the remote machine sees because it is "exporting" a filesystem. If the file-related traffic is too high, then perhaps copies of the files should be distributed to the machines using the files. The other number is the percentage of file requests on the local machine, broken down by filesystem, that are serviced remotely because the local machine is "importing" a filesystem. If that number is too high, then perhaps *it* should be the filesystem's exporter, or perhaps an entirely separate copy of the filesystem should be made.

Another project in the works at Berkeley is a Remote Procedure Call (RPC) system [White] based on the work in [Cooper]. This project will mostly affect the IPC instrumentation, but we don't have any specific suggestions. In fact, a general area for research is the integration of additional protocols (e.g., Xerox's **XNS** protocol) into Berkeley UNIX. As these protocols are built, there must be instrumentation to support them.

One unfortunate side effect of this and other research is that the details given in this report may be out-of-date by the time anyone tries to use them. Once the 4.3BSD distribution tape has been made, all code is liable to being rewritten. Some routines may do different tasks, others may disappear entirely. In fact, we cannot guarantee that the overall organization of the source code will stay the same. Because there is no way to predict these changes, our only advice is to use *tags* (documented under **ctags** in [UserManual]) to find routines and type definitions.

6.3 Test Suites

In this section we shall first present our motivation for test suites. We'll then make some general observations about how the suites should work, and we'll close by suggesting certain suites.

One way to find bugs is to stare at the code and find them "by inspection." We did some of that for this project. Unfortunately, it's often hard work just understanding the simple cases. The obscure, bug-causing cases are even harder to grasp. We don't recommend this technique. A better technique is to run the instrumentation programs on a live system and look for numbers that don't look right. We caught quite a few bugs this way, too. There are two problems with this approach, though. The first problem is that a strange number can result from a bug in the instrumentation, or it can result from a bug in the system being measured. It may even be that you didn't fully understand the system you're measuring. If the system is complex, it can be very hard tracking down the cause of the discrepancy. The second problem is that just because a number looks okay, that doesn't mean it's correct. A reliable but difficult approach is to measure the same values using two completely different methods. The problem with this approach is that many of the measurements we have presented rely on counting; a second independent measuring technique would be very hard to find. Thus we recommend checking out the code by exercising the system in a *controlled* way and checking that the instrumentation numbers agree with your expectations. (Unfortunately, we didn't use this technique at all for this project.)

To get a controlled environment in UNIX, run the system in *single-user mode*. In this environment the only processes that will be running, other than a shell (command interpreter) and your test programs, are the pager and the swapper. The test programs are not the same as the workload generators typically used in performance analysis. Those programs strive for "realistic" loads. All we want are programs (or command files) that cause certain events to happen in a deterministic way. Whether or not the events are realistic is less important.

The appendix of [Leffler *et al* 84] contains short, simple programs that were used for tuning 4.2BSD. These programs provide a model for the ones we have in mind, and some (like the program that sends itself messages) could even be used directly. There are unfortunately too many numbers that we want measured for us to list all of the necessary test programs. Nevertheless, we can make some general observations and a few specific suggestions.

Some IPC-related test programs should generate messages of varying sizes. They should exercise all of the different paths in Berkeley UNIX for sending and receiving messages (e.g., **write**, **send**, **sendto**, **sendmsg**), using all the applicable instrumented protocols. There should be test programs that simulate certain activities of remote hosts. Some of these programs must even *run* on remote machines (e.g., to cause packet collisions on an Ethernet [Shoch & Hupp], [Cabrera *et al*]). You could, for example, probably exercise the mbuf instrumentation adequately by varying the number of routing entries, sockets, and TCP connections in the system.

We primarily want the I/O-related programs to exercise the disk. This means causing seeks, reads, and writes to occur in a controlled way, as well as doing specific numbers of **sync**s and **fsync**s. Also, we want to compare **dumpfs**'s results with the arguments to **newfs** that created the filesystem, and we also want to exercise the instrumentation of slow devices (e.g., tapes, printers). The trick for disk tests will be controlling seeks and transfer rates. This trick will be easier to do if the test is run on a freshly-created filesystem, which you'd get when you test **newfs** and **dumpfs**.

The virtual memory test programs should force certain paging rates (e.g., reclaim rate, rate of dirty page pushes, page scan rate by the pager process) and cause a certain number of swapins and swapouts. One test would be to compile a program and immediately execute it, forcing the system to "attach" pages. Some routines will need to lock pages in core in a controlled way. This must be done inside the kernel.

Exercising the process-related instrumentation involves creating a fixed number of processes. Processes could do things like sleeping, sending themselves signals, and changing their priorities. Another test is to write entries to **/etc/utmp** and make sure that **rwho**, **ruptime**, **uptime**, and **who** all give the right results. Also, we want programs that exercise each of the fields in an accounting record. For example, to test the memory usage field, a program could allocate a predetermined number of bytes. To test the user time field, a program could execute a short loop a few million times (i.e., long enough that the process's user time is essentially the time it spent in the loop). If the execution time of the loop is known, you can compare the user time given by the accounting package with the expected time.

6.4 From the kernel to user-level

UNIX programs running in user mode cannot directly access kernel memory. In 4.2BSD there are therefore two special device drivers. One (**/dev/kmem**) gives user processes access to kernel virtual memory, the other (**/dev/mem**) gives access to physical memory. To read one or more bytes at some location, the process opens the associated special device, seeks to that location, and then reads as many bytes as it wants. Programs like **vmstat** or **ps** use **nlist** to find the kernel virtual address of variables that they want to read. They call **nlist** with a list of strings (e.g., "proc," "rate," "avenrun"), and **nlist** returns a list of addresses.

There are two problems with this approach. The first problem is security. If you can read any location of kernel memory, you can read passwords. The solution to this problem is to make **/dev/kmem** and **/dev/mem** readable only by members of a particular group (the **kmem** group at Berkeley) and allow "authorized" programs to run setgid to that group. Unfortunately, this puts a bureaucratic crimp on users who want to write instrumentation programs that, for example, display pie charts or correlate two different load metrics. The second problem is that most programs must typically read a few variables every time they print results. This raises the number of system calls (one seek and one read per different variable). It also raises the likelihood of not getting a consistent snapshot: the variables might change while the program is pulling out its information.

So let's consider schemes that don't use **/dev/kmem**. One suggestion is to implement an instrumentation system call (e.g., [Eckhardt]). This call would take as an argument a pointer to a buffer and a value telling what variables the program is interested in. This scheme avoids the security loophole that we just mentioned. It can be made atomic because the code can lock the user's pages in core and raise the system interrupt level during the transfer. The chief drawback to this approach is its inflexibility. Sites with binary-only licenses would not be able to change the system call code. They could not, for example, decide to look at **cnt** if that weren't already provided for.

The scheme that we recommend is derived from this idea, though. If we have an "instrumentation device" driver, then we have the advantages of the system call approach, plus binary-only sites can write their own drivers if they so choose. Programs can gather information using **ioctl**s. A sample **ioctl** might be

```
ioctl(instrdev, IIOCGETLOAD, &buf);
```

to copy predefined load numbers (e.g., average number of runnable processes, disk transfer rate, and network transfer rate) into *buf*. This scheme has the same advantages as the system call approach and none of its drawbacks.

We also think that the concept of an "instrumentation server" should be explored. As described in [Kupfer], an instrumentation server would let machines gather instrumentation numbers from remote machines the same way they gather them from the local machine. However, it's not clear whether such an implementation is efficient enough to be the bottom-level instrumentation interface.

6.5 Tools

We think of instrumentation tools as falling into three categories. The first category contains all the tools we have talked about so far: UNIX-flavor "tools." In the second category are picture-oriented programs. Programs of this type already exist at Berkeley, and one called **systat** will be in the 4.3BSD release. The third category contains what we refer to as "integrated" tools. These are programs that coordinate different instrumentation values, such as the load average, the pagein rate, and the disk traffic. Programs in this category also exist at Berkeley. One called **vsta** will be in the 4.3BSD release.

simple tools

We define as UNIX-flavor tools those that produce output that other programs can use. These programs are necessary for three reasons. First, they let us record instrumentation numbers over a long interval. Second, they provide exact numbers when those are desired (as opposed to pictures, which give approximate results). Third, it's still not cheap enough, either in terms of CPU cycles or display hardware, for everyone to see pretty pictures when they want to monitor the system. However, we feel that using, say, **vmstat** output to drive another program is a bad idea except for quick hacks. First, under a heavy load there is no guarantee that **vmstat** will run when it's supposed to unless you give it a hefty boost in priority. Programs that try to correlate information from, say, **vmstat** and **iostat** are doomed to be unreliable. Second, there is additional system overhead from extra context switches and passing characters through a pipe. Finally, there is the added inefficiency of converting a slew of numbers into a readable character form, only to convert them right back into binary. It is usually better just to read the numbers from **/dev/kmem** (or the proposed instrumentation device) directly instead of going through some intermediary.

Nevertheless, if your program really needs to read output from **vmstat**, then a "no headers" option for **vmstat** would be helpful. **Vmstat** and **iostat** both print headers every 20 lines so that you don't forget what the different columns mean. Also, they take pains to squeeze all the numbers into 80 columns, so that everything fits on a standard terminal screen. The "no headers" flag would tell **vmstat** not to print headers, and it would let **vmstat** use as many columns as necessary.

We should point out, though, that programs like **ps** already have (perhaps too) many options. In previous sections of this paper we have suggested more options, and now we've just suggested another. We caution implementors not to blindly add options to programs. In particular, it may be better to split programs like **vmstat** and **iostat** into different programs, rather than overdosing on options.

picture-oriented tools

Another class of tools are those that draw pictures. At the simple end of the spectrum, programs can generate bar charts showing CPU and disk utilization. An example of this type is the Cedar **Watch** tool [Teitelman]). At the sophisticated end of the spectrum, programs can generate a diagram showing how "balanced" the system is (i.e., point out bottlenecks). Such a picture is called a *Kiviat graph* [Ferrari]. A moderately sophisticated program might generate a picture showing the communication between nodes of an internetwork, using the thickness or color of the line to show the traffic rate between any two nodes.

Even standard 24-line by 80-character terminals can display simple pictures via the UNIX **curses** package. Also, there seems to be a trend towards personal UNIX workstations, each having its own high-resolution bit-mapped display. This trend opens up many possibilities for picture-oriented instrumentation tools. The introduction of high-resolution color displays will widen the range of possibilities still further. Our only regret is that different vendors' workstations will probably have different graphics interfaces.

integrated tools

A third class of tools integrates the display of different instrumentation numbers. The program **vsta** attempts to show the state of the major subsystems. The idea is that a user need only run **vsta** to find the system bottleneck, rather than running **ps**, **iostat**, **vmstat**, and **netstat** to track down the problem. One problem in designing **vsta** was deciding on what numbers were absolutely essential and how to fit them all on the screen. Hence, you might want other integrated tools that are more narrowly focused. For example, you might want a program that gives detailed concurrent information about disk traffic and paging rates.

6.6 Software Engineering

In this section we shall present a few specifics about what to do and what not to do when writing instrumentation code. We'll also present a couple of general comments on problems that we've seen in existing code. One problem is the use of floating point. Not every machine running Berkeley UNIX supports floating point numbers efficiently, so fixed point representations should be used when fractions are needed. We also find fault with **printf** format strings that look like `"%5d%4d%5d."` Instead use `" %4d %3d %4d."` Even if the numbers being displayed will "never" spill into the most significant digit, bugs in the kernel or in the instrumentation routine may cause an overflow. It may not be a simple matter to figure out where in the resulting tangle of digits one number ends and the next begins.

Now for the list of things that you *should* do. First of all, use header files to define common data (or even common code if you're sharing a routine between the kernel and a user-level program), unless all the occurences are near each other. It may be easier in the short run just to copy a number or algorithm, but the first time you make a change and forget to propagate it everywhere, you'll wish you'd defined it in a header file. Also, beware of byte-swapping and architectural differences in general. For example, **rwhod** sends packets in binary form. Thus a 68000-based system cannot read **rwhod** packets broadcast from a VAX, which is a ridiculous situation to be in. Finally, work in sanity and consistency checks where possible. For example, if you have a count of mbufs and have information on their allocation, make sure that all mbufs are accounted for. In the accounting package, make sure that the total of user and system times doesn't exceed the elapsed (wall-clock) time.

A final suggestion is to include version stamps in kernel structures, including regular system structures like **proc** and **user** as well as metering structures like **vmmeter**. We at Berkeley find ourselves constantly tripped up by new kernels that break debuggers, **ps**, **w**, and so forth because the kernel has, say, a differently defined u. area. In fact we should consider ourselves lucky--the programs usually break in obvious ways (e.g., the load average always comes out zero or the program blows up). Instead, we should put ourselves in the position that if the header file changes, the user programs will themselves complain that they need to be recompiled.

Of course, there are reasons *not* to use version stamps. One argument against version stamps in structures is that they waste space. On the other hand, the stamps can be conditionally compiled in, so that only sites doing kernel work need include them. Another argument against version stamps is that somebody must maintain them, and they are useless if not kept up-to-date. At least at Berkeley, though, SCCS [Allman] is used to maintain the kernel sources. A scheme using SCCS keywords could keep the version stamp current.

We would now like to tell two sermons. One day while we were working on this paper we encountered an interesting problem. **Vmstat** claimed that a machine was not spending any time in user mode, any time in system mode, or any time idle. Because this was a research machine, our first reaction was to recompile **vmstat**. The problem didn't go away. After poking around in the kernel for awhile, we decided that the machine had missed an interrupt from the profiling clock, which must be "reprimed" after each interrupt. That is, at some point the clock went off, but its interrupt handler didn't get called, so the clock didn't get reprimed, so we never heard from it again. We rebooted the machine to get everything back to normal. The most obvious moral of this story is that the sampling code isn't as robust as we'd like, though it's not clear how we'd fix it. The other moral is that the system is always capable of surprising you.

The second sermon concerns hidden assumptions in code. If we look at the definition of the **vmmeter** structure, we find the two fields **v_first** and **v_last**. If we look in the code for the **vmmeter** function, we find that it expects that the members beginning with **v_first** and ending with **v_last** are all normal-size integers (**ints**). If someone slipped a **short** integer or a floating point number in between **v_first** and **v_last**, the code would break. Now, this assumption of all **int**'s is obvious from looking at the code. But it would not be obvious just from looking at the definition of the **vmmeter** structure, nor would it be obvious from reading our description of how the **vmmeter** routine works. Two morals here: "Get rid of the hidden assumptions!" and "Watch Your Step."

6.7 Documentation

Documentation at a research organization is rarely up-to-date. Even if you're willing to expend the energy to document each and every change when it happens, you still find yourself unable to keep up with all the changes. So it was no surprise when we found that the on-line documentation described a world noticeably different from the one described by the code. In fact, for this paper we gave up noting each and every documentation bug. It seemed more profitable just to wait until all the code is frozen for the next release and then go in and revamp the documentation. Nevertheless, we can at least point out a couple of trouble areas. Some of the manual pages still cling to the belief that "the" UNIX clock runs at 60 Hz. As we have mentioned, that is no longer true at Berkeley; neither clock runs at 60 Hz. Also, the manual pages for **pstat** and **ps** explain the meanings of the various flags that might be displayed. Unfortunately, many of the flags have changed meaning since the manual pages were written.

We would also like to see more complete documentation. For example, the manual page for **ps** tells us that **RSS** is the "real memory (resident set) size of the process." From that you might assume that it includes all three segments: text, data, and stack. Or you might assume that it includes only data and stack, as the **p_rssize** member of a process table entry includes only those segments. But no, the answer is that **RSS** is the sum of the data and stack segments, plus the process's "share" of the text segment if the text is shared. This is a reasonable approach, as it shows the load the process is placing on physical memory. We just wish that **RSS** were more precisely defined. Another example is the documentation for **netstat**. It is likely that the people who write distributed programs will know little about the details of, say, TCP. However, when they use **netstat** to debug their programs, they'll be expected to know what **FIN_WAIT_2** means. That sort of information should be in the **netstat** documentation.

7. Conclusions

After poring through most of the Berkeley UNIX kernel and looking at almost every "official" instrumentation program, we would like to make two observations. First of all, the 4.2BSD instrumentation for IPC is clearly skeletal. This is of course understandable, as it is new and still being developed. Second, Berkeley UNIX instrumentation faces a large danger from software rot. For example, we spent a lot of time tracking down bugs in **pstat**, a program that used to work. It's even likely that some of the bugs won't get fixed until after 4.3BSD is released because of time constraints. **Netstat** is another example: it sprang a leak when a new type of mbuf was created. We think that this points to a need for better engineered code: code that will track future changes and that will not fail in subtle ways. We note that test suites ironically provide yet another place for software rot to strike, but we think that they, along with many of the other ideas discussed in this paper, will be worthwhile in the long run.

Acknowledgements

Golly, this project's been a lot of work. But it wouldn't have been possible without the support and guidance of Luis Felipe Cabrera and Domenico Ferrari. The project would also have been sunk were it not for the patient assistance of Mike Karels, who was always willing to answer "just one more question." I also thank Kirk McKusick for the questions that he answered.

I got some ideas for this project from talking to other Berkeley students, I got a couple of ideas from talking to other summer interns at Xerox PARC, and I got lots of ideas from people who I talked with while job-hunting. Not all of those ideas made it into this report, but I'd like to thank Chris Baldwin, Dave Goldberg, Rusty Sandberg, and Gerd Tuchen for their suggestions that did make it in. Also, thanks to Ed James for describing his quota-management tool.

I'm grateful to the organizations that provided my financial support for the last two years. I'd also like to thank Keith Sklower and Diana Wear for their help with the mechanics of getting the report written. Also, thanks to Robbie Rackar for reading an early draft of the report. Last, but not least, thanks to Miriam Amos for her support and for not letting me goof off (much).

References

[Allman]
Eric Allman. "An Introduction to the Source Code Control System," *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

[Birrell & Nelson]
Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984.

[Cabrera *et al*]
Luis Felipe Cabrera, Edward Hunter, Mike Karels, and David Mosher. "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD," Berkeley Technical Report UCB/CSD 84/217, December 1984.

[Cooper]
Eric C. Cooper. "Circus: A Replicated Procedure Call Facility," *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, October 1984.

[Eckhardt]
Dave Eckhardt. USENET article <1587@psuvax1.UUCP>, Pennsylvania State University, December 1984.

[Ferrari]
Domenico Ferrari. *Computer Systems Performance Evaluation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Graham *et al*]
Susan L. Graham, Peter B. Kessler, and M. Kirk McKusick. "Gprof: A Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 17, No. 6, June 1982.

[Gusella & Zatti]
Riccardo Gusella and Stefano Zatti. "TEMPO: Time Services for the Berkeley Local Network," Berkeley Technical Report UCB/CSD 83/163, December 1983.

[Hunter]
Edward Hunter. "Adding Remote File Access to Berkeley UNIX 4.2BSD Through Remote Mount," M.S. Report, University of California, Berkeley, December 1984.

[ICMP]
"Internet Control Message Protocol," RFC 792, Information Sciences Institute, Marina del Rey, California, September 1981.

[IP]
"Internet Protocol," RFC 791, Information Sciences Institute, Marina del Rey, California, September 1981.

[Kernighan & Ritchie]
Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Kridle]
Bob Kridle. Private communication, 14 May 1985.

[Kupfer]
Michael Kupfer. "Performance of a Remote Instrumentation Program," Berkeley Technical Report UCB/CSD 85/223, February 1985.

[Leffler]
Samuel J. Leffler. "Building 4.2BSD UNIX Systems with Config," Computer Systems Research Group, University of California, Berkeley, July 1983.

[Leffler *et al* 83a]
Samuel J. Leffler, Robert S. Fabry, and William N. Joy. "A 4.2BSD Interprocess Communication Primer," Berkeley Technical Report UCB/CSD 83/145, July 1983.

[Leffler *et al* 83b]
Samuel J. Leffler, William N. Joy, and Robert S. Fabry. "4.2BSD Networking Implementation Notes," Berkeley Technical Report UCB/CSD 83/146, July 1983.

[Leffler *et al* 84]
Sam Leffler, Mike Karels, and M. Kirk McKusick. "Measuring and Improving the Performance of 4.2BSD," Berkeley Technical Report UCB/CSD 83/218, May 1984.

[Macrander]
Cathryn M. Macrander. "Development of a Control Process for the Berkeley UNIX Distributed Programs Monitor," M.S. Report, University of California, Berkeley, December 1984.

[Malis]
Andrew G. Malis. "The ARPANET 1822L Host Access Protocol," RFC 878, Information Sciences Institute, Marina del Rey, California, December 1983.

[McKusick *et al* 84]
M. Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984.

[McKusick *et al* 85]
M. Kirk McKusick, Mike Karels, and Sam Leffler. "Performance Improvements and Functional Enhancements in 4.3BSD," *Proceedings of the Portland Usenix Conference*, June 1985.

[Opperman & Davis]
Mark R. Opperman and Mike B. Davis. "4.2BSD UNIX File System Performance," unpublished CS 262 class report, University of California, Berkeley, Spring 1984.

[Ousterhout *et al*]
John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. "A Trace-Driven Analysis of the UNIX 4.2BSD File System," Berkeley Technical Report UCB/CSD 85/230, April 1985.

[Plummer]
David C. Plummer. "An Ethernet Address Resolution Protocol," RFC 826, Information Sciences Institute, Marina del Rey, California, November 1982.

[ProgrammerManual]
UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

[Ritchie]
Dennis M. Ritchie. "The UNIX I/O System," *UNIX Programmer's Manual*, Seventh Edition, Volume 2B, 1979.

[Ritchie & Thompson]
Dennis M. Ritchie and Ken Thompson. "The UNIX Time-Sharing System," *Communications of the ACM*, Vol. 17, No. 7, July 1974.

[Saltzer & Gintell]
Jerome H. Saltzer and John W. Gintell. "The Instrumentation of Multics," *Communications of the ACM*, Vol. 13, No. 8, August 1970.

[Sechrest]
Stuart Sechrest. "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2bsd," Berkeley Technical Report UCB/CSD 84/191, June 1984.

[Seymour & Lob]
Harlan Seymour and Clifford Lob. "UNIX 4.2BSD File System Locality Measurements," unpublished CS 262 class report, University of California, Spring 1984.

[Shoch & Hupp]
John F. Shoch and Jon A. Hupp. "Measured Performance of an Ethernet Local Network," *Communications of the ACM*, Vol. 23, No. 12, December 1980.

[Tanenbaum]
Andrew S. Tanenbaum. *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Teitelman]
Warren Teitelman. "The Cedar Programming Environment: A Midterm Report and Examination," Xerox Palo Alto Research Center Report CSL-83-11, June 1984.

[Thompson]
Ken Thompson. "The Unix Time-Sharing System: Unix Implementation," *Bell System Technical Journal*, Vol 57, No. 6, July-August 1978.

[TCP]
"Transmission Control Protocol," RFC 793, Information Sciences Institute, Marina del Rey, California, September 1981.

[UDP]
"User Datagram Protocol," RFC 768, Information Sciences Institute, Marina del Rey, California, August 1980.

[Ultrix]
ULTRIX-32 Documentation Set, Digital Equipment Corporation, May 1984.

[UserManual]
UNIX User's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

[White]
Karen White. Untitled M.S. Report, University of California, Berkeley, in preparation.

Appendix

This Appendix summarizes the routines that we looked at for this project. The listing for each routine tells what sections it was discussed in and what bugs we found in that code. The problems listed here are not our only complaints, just the ones that cause the code not to do what it's advertised as doing. Not all of these bugs were mentioned in the report. Not all of these bugs are necessarily in the 4.2BSD release, for reasons discussed in the report. Bugs that have been fixed after we discovered them are so noted.

accounting (Sections 5.2 and 5.3)

Many kernel routines check **u.u_uid** themselves, rather than going through **suser**. In particular, the following functions are guilty of this: **fork** (but only the second call; the first call is just to save some work), **killpg1**, **chkdq**, **chkiq**, **ttioctl** (all 4 calls), **ptsopen**, **alloc**, **realloccg**, **access**, **namei**, **socreate**, **in_pcbbin**, **cnopen**, **dhopen**, **dmfopen**, **dzopen**, **setpgrp**, **donice**, **kill**, **chmod1**.

The billing of **ru_oublock** for delayed-write buffers (in **realloccg**, **bwrite**, and **bdwrite**) is only guaranteed to be right if the same process does the the second write.

The data segment gets billed for the process's entire resident set size; the stack segment doesn't get billed for anything.

df (Sections 3.2 and 3.3)

dumpfs (Sections 3.2 and 3.3)

iostat (Sections 3.2 and 3.3)

iostat uses **sleep** instead of **alarm** and **sigpause**.

kernel (excluding accounting) (Sections 2.1, 2.2, 2.3, 3.1, 3.2, 3.3, 4.1, 4.2, 4.3, 5.1, 5.2, and 5.3)

The "collisions" count for the IMP interface has nothing to do with packet collisions.

Code to support **vmstat -t** (timing information about page faults and reclaims) is broken.

schedcpu doesn't know when it's been called late.

vmmeter uses "if (time.tv__sec % 5 = = 0)" instead of a counter to decide when to call **vmtotal**.

schedcpu calls **timeout** just before it returns, rather than right after it's been called.

sbreserve had the line "`sb->sb_max = MAX(cc * 2, SB_MAX);`". The call to **MAX** should have been to **MIN**. FIXED.

DK__NDRIVE is fixed at 4, rather than actually being the number of drives in the system.

The type **off__t** should be an **unsigned long** instead of an **int**.

The **t__col** field (in **struct tty**) should be an **unsigned char**.

nswdev hasn't been set up by the time **vminit** is called. So the test "`if (nswdev >= 2)`" always fails, so **maxpgio** is usually two-thirds of what it should be.

netstat (Sections 2.2 and 2.3)

netstat -m claimed that the network memory in use was a negative percentage of the network memory allocated. FIXED.

netstat -m claimed that there were missing mbufs. This happened when we created a new mbuf type. FIXED.

netstat -p is not implemented.

netstat doesn't actually decide which interface is the busiest in, say, "netstat 5".

netstat -I uses **sleep** instead of **alarm** and **sigpause**.

ps (Sections 2.3, 3.3, 4.2, 4.3, 5.2, and 5.3)

ps thinks that sleeping at **PZERO** is interruptible.

ps checked **SWEXIT** before checking **SZOMB**, so it never displayed "<defunct>". FIXED.

pstat (Sections 3.2, 3.3, 4.2, 4.3, 5.2, and 5.3)

pstat -t assumed that there would never be more than 128 DZ or DH (terminal) lines on any one system. FIXED.

pstat -s was reporting missing swap space. This was because calls to **clrnd** and **ctod** were missing. FIXED.

pstat -u used **/dev/kmem** instead of **/dev/mem**. FIXED.

pstat -u printed one field twice and didn't print all the system call arguments. FIXED.

pstat -u is broken because `sizeof(struct user) > CLBYTES`.

pstat -p printed **CLKT** in the banner but didn't print a number in that spot. FIXED. (The number that used to get printed there is no longer kept where **pstat** can get at it without going through more bother than it's worth.)

The **ADDR** field given by **pstat -p** was wrong. FIXED.

rwho (Sections 5.2 and 5.3)

Ruptime and **rwho** don't agree on long to wait before declaring a system to be down. FIXED (but not cleanly).

rwhod (Sections 5.2 and 5.3)

Rwhod packets are in binary, so machines with incompatible architectures can't read each others' packets.

ruptime (Sections 5.2 and 5.3)

Ruptime and **rwho** don't agree on long to wait before declaring a system to be down. FIXED (but not cleanly).

vmstat (Sections 3.2, 3.3, 4.2, and 4.3)

vmstat uses **sleep** instead of **alarm** and **sigpause**.

who (Sections 5.2 and 5.3)

w/uptime (Sections 5.2 and 5.3)