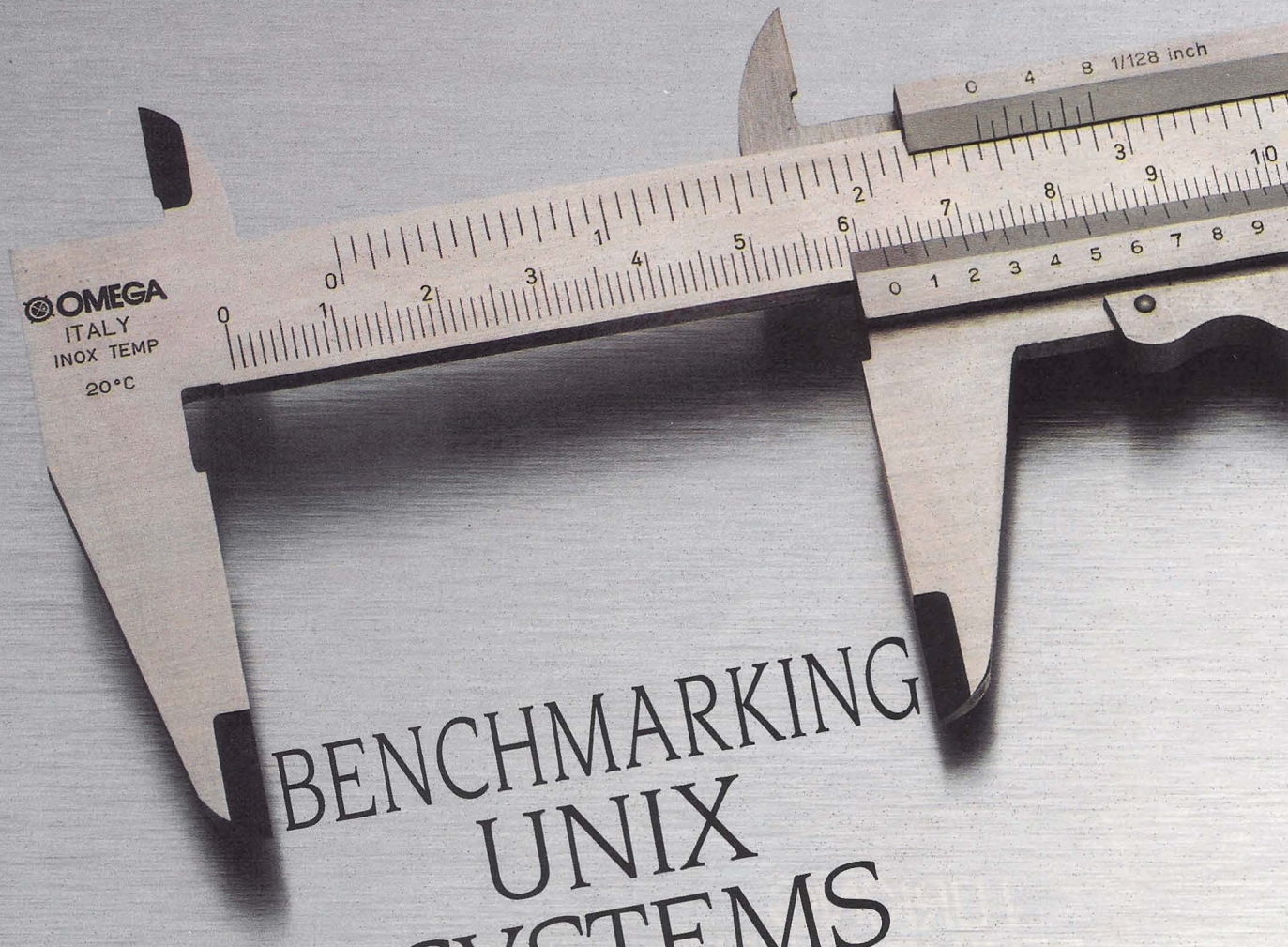
UNIX is a trademark of Bell Laboratories.



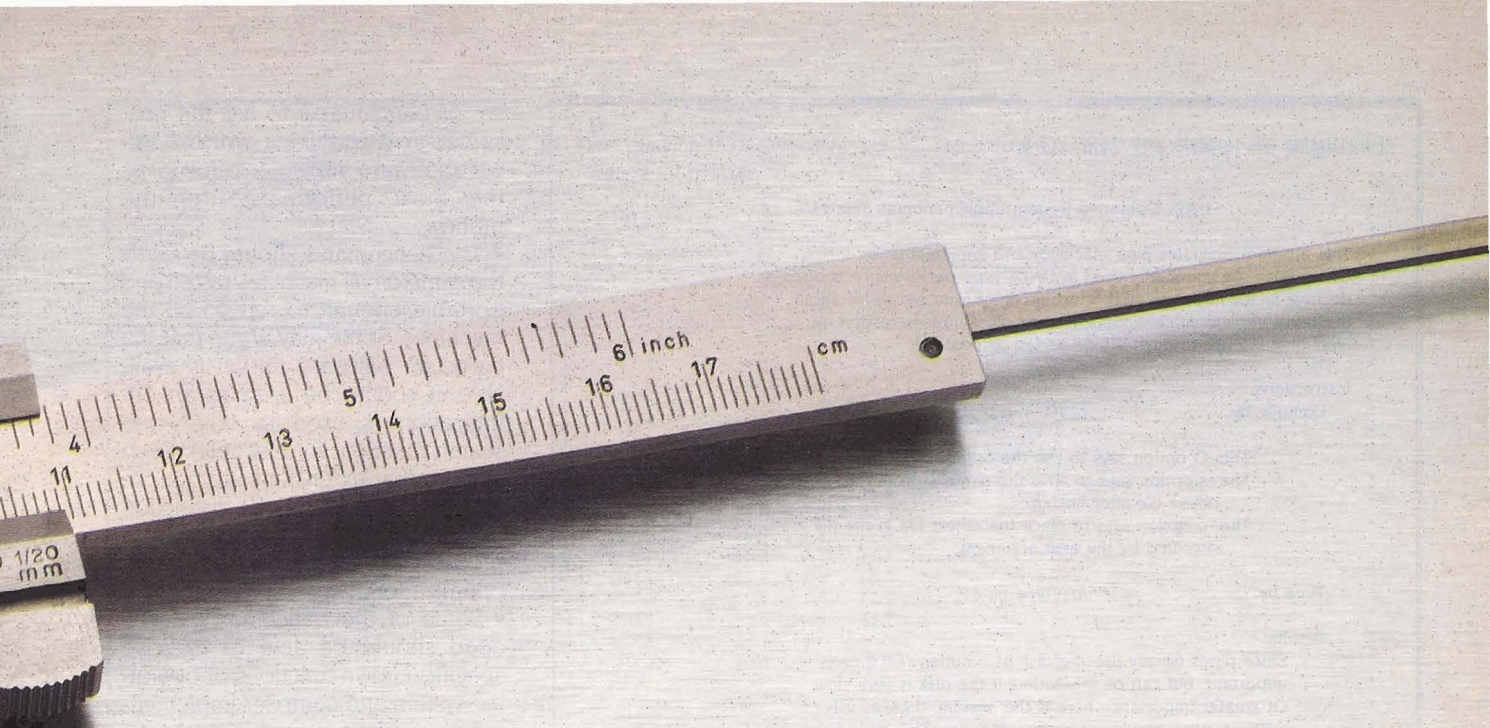
BENCHMARKING UNIX SYSTEMS

UNIX *performance on microcomputers and minicomputers*

BY DAVID F. HINNANT

WITH THE ADVENT of inexpensive but powerful 16- and 32-bit microprocessors, the multiuser, multitasking UNIX operating system is available on many new microcomputers. Because UNIX is becoming a standard multiuser operating system, this article presents benchmarks for several microcomputer UNIX implementations and shows how they perform when compared to minicomputer versions of UNIX.

Almost every week, new versions of UNIX for popular microcomputers or new UNIX-based microcomputers are being announced. They come from everyone from AT&T (UNIX was developed at Bell Laboratories) and IBM down to previously unknown companies who license UNIX from AT&T and then implement, or port, it for new hardware configurations. For some UNIX implementations on microcomputers, you need additional hardware (at least a hard-disk system for the megabytes of



utilities and programs that come with UNIX), while some implementations run by just inserting a disk and booting up the run-time operating system.

Quite a few releases of UNIX software are also available, including Version 6, Version 7, System III, and System V (AT&T's new standard) from Bell Labs; the 4.1 BSD and 4.2 BSD UNIX extensions from the University of California, Berkeley; XENIX, an enhanced UNIX Version 7 from Microsoft; VENIX, a finely tuned UNIX with real-time capabilities; Uniplus+, an enhanced UNIX System III from Unisoft; also, many systems claim to be "UNIX-like." Do all these UNIX implementations look and work alike? What do you look for in determining which is the better UNIX-based machine? Some will be faster than others because of hardware constraints, but how can you tell if you're getting the best system for your operating conditions? That's usually the bottom line. Enter the UNIX benchmarks.

Just what is a benchmark? For our purposes, a benchmark is a set of instructions that measures how well the hardware and software of a computer system perform together. Benchmarks can either exercise singular, specific functions of a compiler or operating system (e.g., function-call overhead or system-call overhead) or they can test the general performance of the machine by exercising a number of operations (e.g., looping, searching, etc.). In both cases, these timings are compared to results on other machines.

Benchmarking an operating system is much more complex than benchmarking a compiler. However, since UNIX and its language compilers are written in the high-level language C, compiler implementation is just as important, as we will see. What should an operating-system benchmark do?

The purist approach tests only well-defined individual functions of the operating system (e.g., pipe implementation, disk throughput, system-call overhead, and context switching). This is difficult to do, as any program that exercises one particular operating-system function inadvertently includes statements that are not related to the function under scrutiny. We must reconcile ourselves to a benchmark that includes as little superfluous code as possible. Since this unwanted overhead varies from program to program, in most cases, the execution-time results cannot be manipulated in an arithmetic fashion to produce true timings of a particular operating-system function. They can, however, be compared to timings on other machines.

The practical approach tests overall system performance by performing many typical functions (e.g., sorts; compilations; and creating, listing, and deleting files) but in a controlled fashion. Again, these timings are compared with timings on other UNIX machines.

Both approaches have merit, and examples of both will be presented here.

Some Benchmark Guidelines

Let's develop some guidelines for benchmarking operating systems in general and apply them specifically to UNIX. In order to benchmark any operating system, as many environmental variables as possible should be eliminated:

- The benchmarks should be run in a multiuser environment so that any normal operating-system overhead (e.g., context switching between processes and other operating-system housekeeping) is taken into account. (Most UNIX implementations offer a single-user mode as well as a multiuser mode.)
- The benchmarks should be taken more than once and averaged to compensate for any system-level background processes that repeatedly sleep awhile and work awhile. (A good UNIX example is the routine update that wakes up every 30 seconds and, among other things, flushes buffers and in-core tables to disk.)
- The benchmarks should be performed while the system is otherwise idle so that the timings are as true as possible. On microcomputers, even one additional user can distort benchmark results significantly; depending, of course, on what is being done.
- All available optimizations should be used. Implementations that take advantage of the hardware con-

(continued)

David Hinnant (2017 Hunterfield Lane, Raleigh, NC 27609) holds a B.S. degree in physics and is a UNIX systems programmer with ITT Telecom in Raleigh. He is coauthor of a book on UNIX microcomputers soon to be published by Robert J. Brady Co. His UNIX UUCP address is ...ucbvax!decvax!ittvax!ittral!hinnant. Machine-readable copies of the benchmark suite are available upon request to the UUCP address. Benchmark results from UNIX systems not mentioned in this article are welcome.

PHOTOGRAPHED BY PAUL AVIS

Listing 1: *The UNIX pipe benchmark.*

```
/*
 *                  UNIX Operating System Implementation Test #1
 *
 *  This program evaluates pipe efficiency and implementation.
 *  Since pipes are commonly used in UNIX, pipe performance is often a
 *  decisive factor in overall system performance, and says a lot about
 *  the UNIX implementation. Here we test pipe implementation by
 *  cramming 0.5 MB through a pipe as fast as possible.
 *
 *  Instructions:
 *     Compile by:              cc -O -s -o pipes pipes.c
 *
 *              The -O option says to use the optimizer.
 *              The -s option says to strip the namelist from the
 *                  object file after linking.
 *              The -o option says to place the object file in the file
 *                  specified by the next argument.
 *
 *     Time by:                 /bin/time pipes
 *
 *     Results:
 *          Since pipes usually use the disk as a buffer, real time is
 *          important, but can be misleading if the disk is very slow.
 *          Of greater importance here is the 'system' time, as it is
 *          a direct measurement of kernel efficiency. The 'user' time
 *          is of little importance.
 *
 */

#define BLOCKS 1024

        /* the buffer */
char buffer[512];
        /* file descriptor for pipe */
int fid[2];

main()
{
                /* want to test pipe implementation; not arithmetic */
        register int i;
                /* initialize the pipe */
        pipe(fid);
                /* fork the child process */
        if ( fork() ) {
                        /* parent process writes to pipe in 512 byte chunks */
                for (i = 0; i < BLOCKS; i++)
                        if (write(fid[1], buffer, 512) < 0)
                                        /* if there is a problem, say so */
                                printf("Error in writing; i=%d\n", i);
                        /* close the pipe when we're done */
                if (close(fid[1]) != 0)
                        printf("Error in parent closing\n");
        }
        else {
                        /* close, since we aren't writing */
                if (close(fid[1]) != 0)
                        printf("Error in child closing\n");
                        /* child process reads the pipe until EOF */
                for (;;)
                        if (read(fid[0], buffer, 512) == 0) {
                                break;
                        }
        }
}
```

figuration are generally better to begin with. If a compiler option for object-code optimization is available, it should be used. If the hardware can support fast (register in the case of C) variables, they should be used. In the benchmarks discussed here, all variables that can be of the register type will be declared as such. In reality, the number of registers available for use by register variables varies widely because of hardware differences between microprocessors. Remember, our goal here is not to develop benchmarks that determine which UNIX machine is the best under a given set of requirements but to develop a general set of benchmarks to aid the consumer in determining which hardware/software implementation gives the most performance for the money.

- The benchmarks should be *exactly* the same on all machines tested and portable enough to run on all the machines. Some may argue that if a particular software option is available, it should be used as an optimization is used (a binary-tree search function, for example). Keep in mind that extensions are not optimizations. Although the distinction can become cloudy, an extension is probably not used as routinely as an optimization.
- Some of the benchmarks developed should be able to exercise specific, known functions of operating-system and compiler implementation.
- The benchmarks developed should also contain tests of overall performance by simulating typical user activities. This should include executing background processes concurrently with foreground processes (if possible) to see how the system responds under a multitasking load.
- The benchmark timings should be made using a consistent and accurate method. A stopwatch just won't do. Fortunately, UNIX has a standard timing mechanism that reports elapsed (real) and processor times used by a process. The processor time is further divided into user and system times.

User time is the amount of time the process spent executing nonprivileged instructions (e.g., arithmetic calculations, sorting, searching, calling user-level functions, etc.).

System time is the time the process spent executing privileged (kernel) commands (i.e., system calls) plus some system-level overhead (e.g, context switching between processes).

The elapsed time is just that. And it is often not the sum of the user and system times. The majority of the missing time is spent waiting for I/O (input/output) operations to complete, waiting for a signal from another process, sleeping, or swapped out on disk while another program is running. It is unfortunate that in some implementations of UNIX the elapsed time reported by this timing mechanism is given only to the second. Thus, the sum of the system and user times can on occasion be greater

Table 1: *The results of* UNIX *benchmarks for some common microcomputers and minicomputers. The table is sorted on the fastest execution (real) time for the shell benchmark in listing 6a.*

System			Time in Seconds								
			1. Pipe			2. System Call			3. Function Call		
No.	Machine	Version	real	user	sys	real	user	sys	real		
1	VAX-11/780	4.1 BSD	3.2	0.1	1.2	4.8	1.4	4.0	1.0		
2	Masscomp	Sys III+	5.7	0.0	2.8	6.3	0.4	5.8	0.9		
3	Sun-2/120	4.2 BSD	7.6	0.1	3.7	6.8	1.1	5.6	0.8		
4	VAX-11/750	4.1 BSD	4.6	0.2	2.1	7.0	0.9	6.2	1.7		
5	PDP-11/70	2.8 BSD	8.1	0.0	3.4	8.0	0.2	7.5	1.0		
6	Altos 986	XENIX	6.0	0.1	2.8	11.0	0.8	10.3	0.4		
7	IBM PC XT	PC/IX	16.6	0.1	7.6	39.8	2.9	35.6	4.7		
8	PDP-11/23	VENIX	30.0	0.1	9.5	24.0	3.2	20.4	3.3		
9	IBM PC XT %	VENIX/86	18.0	0.1	7.3	20.5	2.3	17.8	2.8		
10	SCI-1000 ^	Sys III+	9.3	0.0	3.1	26.2	0.7	24.2	1.2		
11	Omnibyte	Idris *8:	32.0	0.1	30.4	21.3	2.5	18.4	1.7		
12	TRS-80 16B	XENIX	8.0	0.1	3.4	15.0	1.5	12.7	1.4		
13	PDP-11/23	V7	23.0	0.1	10.7	36.5	0.9	33.7	3.6		
14	DEC Pro/350	VENIX	26.0	0.5	13.8	33.3	5.8	26.5	3.5		
15	Apple Lisa	Sys III+	8.1	0.0	3.0	10.5	0.2	9.1	1.3		

System			Time in Seconds											
			4. Sieve			5a. Disk Write	5b. Disk Read	6a. Shell			7. Loop			
No.	Machine	Version	real	user	sys	real	real	real	user	sys	real	user	sys	
1	VAX-11/780	4.1 BSD	1.7	1.5	0.1	2.0	8.0	3.3	0.3	1.3	2.6	2.5	0.1	
2	Masscomp	Sys III+	2.8	2.5	0.1	1.7	–	3.5	0.4	1.4	6.6	6.3	0.1	
3	Sun-2/120	4.2 BSD	5.1	2.8	0.4	1.8	4.9	3.5	0.3	2.0	7.4	7.0	0.1	
4	VAX-11/750	4.1 BSD	2.4	2.7	0.1	3.0	8.0	3.8	0.4	1.5	5.1	4.9	0.1	
5	PDP-11/70	2.8 BSD	2.3	1.6	0.1	4.0	9.5	4.0	0.2	1.7	7.9	7.1	0.2	
6	Altos 986	XENIX	3.3	3.0	0.0	3.5	7.3	7.0	0.4	1.6	13.3	13.0	0.1	
7	IBM PC XT	PC/IX	8.2	7.8	0.3	11.6	20.7	8.5	1.1	3.2	32.2	31.5	0.3	
8	PDP-11/23	VENIX	5.5	5.1	0.1	8.0	33.7	12.0	0.7	4.8	26.0	25.2	0.1	
9	IBM PC XT %	VENIX/86	9.0	8.2	0.3	7.0	25.6	13.0	0.8	4.2	32.7	31.4	0.3	
10	SCI-1000 ^	Sys III+	4.4	3.6	0.1	4.3	9.1	13.6	0.5	1.9	14.5	13.6	0.2	
11	Omnibyte	Idris *8:	7.0	5.4	0.4	12.3	$	17.6	0.3	16.1	17.0	16.1	0.4	
12	TRS-80 16B	XENIX	6.0	4.8	0.3	8.0	22.0	18.0	0.4	2.6	14.0	12.5	0.5	
13	PDP-11/23	V7	5.8	5.3	0.1	22.0	32.7	20.4	0.8	8.5	27.4	25.9	0.3	
14	DEC Pro/350	VENIX	6.3	5.1	0.1	7.7	28.0	27.0	0.8	4.7	26.7	25.3	0.1	
15	Apple Lisa	Sys III+	6.1	5.3	0.1	20.8	44.5	37.6	0.4	3.2	14.0	12.0	0.2	

\+ Indicates UNIX System III plus some Berkeley enhancements.
* The benchmark in listing 1 had to be modified slightly to run under Idris 2.1, perhaps explaining the large times that resulted.
$ Idris 2.1 is a Version 6-based UNIX system, and hence did not have the **rand()** system call. Thus, the benchmark could not be run.
– Unfortunately, this time was not available at the time of publication.
^ The SCI-1000 benchmarked was a preproduction 80186 system with debugging code in the kernel and compiler.
% For some reason, the C compiler optimizer caused the operating system to crash, so these results are with nonoptimized benchmarks.

System Configuration:

1 — 4-megabyte RAM, two 256-megabyte disk drives
2 — 2-megabyte RAM, one 50-megabyte disk drive
3 — 2-megabyte RAM, one 42-megabyte disk drive
4 — 2-megabyte RAM, one 121-megabyte disk drive
5 — 1.5-megabyte RAM, 400 megabytes of disk drives
6 — 1-megabyte RAM, one 40-megabyte disk drive
7 — 512K-byte RAM, one 10-megabyte disk drive
8 — 256K-byte RAM, two 5-megabyte disk drives
9 — 512K-byte RAM, one 40-megabyte disk drive
10 — 640K-byte RAM, one 10-megabyte disk drive
11 — 384K-byte RAM, one 20-megabyte disk drive
12 — 384K-byte RAM, one 15-megabyte disk drive
13 — 256K-byte RAM, two 10-megabyte disk drives
14 — 256K-byte RAM, one 5-megabyte disk drive
15 — 1-megabyte RAM, one 5-megabyte disk drive

than the elapsed time.

This mechanism is the **time** command, which is invoked explicitly by

/bin/time *filename*

where *filename* is the program to be timed. Under UNIX, *filename* can be either an object file or a text file of shell commands. Of course, some overhead is in the **time** command itself, since it has to start *filename* executing, but it is small and can be neglected because all our benchmarks will be timed this way. The results are compared to other UNIX machines timed in the same manner.

The UNIX Benchmark Suite

How can we apply these guidelines to the UNIX operating system and its most important language, the C compiler? What should we test? That question can be answered by answering the question, "What does UNIX do most often?"

UNIX has a number of unique and powerful features that are used quite heavily. If implemented efficiently, these

(*continued on page* 400)

(continued from page 135)
features can make slow hardware seem fast. If implemented poorly, they can make even the most elegant hardware seem archaic. UNIX benchmarks should concentrate on some critical areas.

UNIX was developed on a small machine with limited memory and is disk intensive by its very nature. Therefore, we should test features of UNIX that use the disk.

The user interface to the UNIX system is called the shell (several common varieties exist). Since all requests made by the user are processed by the shell, it should be tested extensively.

The UNIX pipe qualifies on all the above criteria. A pipe is an I/O channel that is written into by one program and read by another. Pipes are used by a number of UNIX utilities, the shell in particular. Pipes are also often buffered on disk. A UNIX benchmark using a pipe is given in listing 1. The program creates a child process to read the pipe using the **fork()** system call and then crams 0.5 megabyte through the pipe. What do the results tell us? The two times of interest are the system and elapsed times. The system time, for all practical purposes, is a measurement of how long it took to set up and perform the piping. It thus is a direct measurement of pipe efficiency. The elapsed time is of interest because it helps give a good measurement of how slow the disk is. Elapsed time minus system time minus user time is essentially the disk-overhead time. Since microcomputers usually don't have the fastest disks, this is an important measurement for them. User time by itself is of little importance.

So you can get some idea of the time required to execute this and other benchmarks discussed in this article, table 1 shows the timings for some common minicomputers and microcomputers running UNIX. These times are average times on an otherwise idle system, as per the guidelines established above.

In the pipe benchmark, we measured the time it took to perform certain system calls (**fork()**, **read()**, **write()**, etc.) that were related to pipe implementation. The time to perform just one system call can be divided into several components:

1. The time required for the user-program system-call library interface to set up and execute a trap (an SVC to IBM 370 users) to the kernel so that privileged instructions can be executed. When this happens, the registers needed by the processor (stack pointers, program counter, etc.) to run the user program are saved so that they can be restored after the system call is complete.
2. The time the processor is performing the desired function.
3. The time required for the user-program registers to be restored and control transferred so that the user program can resume computation with the result from the system call in hand.
4. The time used when a context switch between processes is required.

It would be nice to measure 1, 3, and 4, since they can be considered the majority of the overhead in making a system call. The program in listing 2 does just that. It does nothing but repeatedly (25,000 times) query the operating system concerning its process identity with the **getpid()** system call. This information is kept in an in-core process table, so access is extremely fast and actual computation very small, as long as no other processes are competing for the processor. (See the need for an idle system?) Since we're interested in measuring overhead, and the program doesn't do much other than system calls, the elapsed time is important here. System time should be close to the elapsed time, and user time should be very small. Both are insignificant. Again, the results of this benchmark are shown in table 1.

Now that we've benchmarked system-call overhead, the overhead involved in an ordinary user function call and return naturally follows. This benchmark may initially seem superficial but consider that it is compiler implementation that to a large degree determines object-code efficiency, and the same compiler (C in our case) is probably used to compile the operating-system kernel. If so, it *should* be considered when evaluating the operating system. It should also be noted that an inefficient compiler can nullify any speed gained by structured-programming techniques. Benchmarking compilers is a topic by itself and will be left alone here. Let's just measure function-call overhead and consider it representative of compiler efficiency.

It is possible to determine the overhead involved in a function call in a number of ways. The method used here is believed to be more accurate than others. Since our comparison is two-way, two programs should be written: one that uses a function to achieve a goal and one that does not. The two programs, however, should perform the same task. After these programs are run, the user-execution time from the program not using the function is subtracted from the user-execution time of the program that does. This difference is the function-call overhead involved. This number can be divided by the number of times the call was made to arrive at a seconds-per-call overhead value, which can be enlightening when compared from system to system. An example of how this is done is shown in listing 3. Even though the program could have been made simpler by not passing a value to the function **empty()**, in real life all functions return at least one value, whether examined or not, and most functions pass at least one value (which is overhead, really). Using the C preprocessor, it is possible to write two distinct programs in one text file, depending on how the text file is compiled. The program in listing 3 is either compiled with **-DEMPTY** to generate the empty function program or with **-DASSIGN** to generate the program that doesn't use a function but achieves the same goal.

As mentioned above, the user time, not the real time, is used in the calculation. This is because the real time is accurate only to the second, whereas the user time is accurate to the tenth. And, since we're generating a nonrelative numerical result, where virtually no system time is used, the measurement with the greater precision is needed.

Let's turn our attention to the C compiler. When most people think of compiler benchmarks, they think of the Sieve of Eratosthenes, which tests compiler efficiency and processor throughput quite well. It's an excellent test for looping, testing, and incrementing. The program in listing 4 is a slightly modified copy of the Sieve presented in the January 1983 BYTE (page 283). Since we're not using a stopwatch, all unnec-

essary I/O has been removed. Also, by the guidelines established above, register declarations have been added. The time to be interested in here is the elapsed time. The user time should be about the same as the elapsed time, while the system time should be quite small.

We briefly touched on disk performance with the pipes test, but disk performance deserves a more in-depth evaluation. UNIX provides methods for both sequential and random-access files, and both should be tested. Listings 5a and 5b are benchmarks that test random-access disk implementation. The program in listing 5a creates, opens, and writes a 256- by 512-byte file. The number of blocks manipulated is specified by a **#define** statement and can easily be changed if it is too large for a small microcomputer implementation. The program in listing 5b randomly reads the file created in listing 5a and removes it afterward.

While sequential access should be tested, it is not presented here since disk access is by and large random access. It should be easy to derive a sequential-access test from the random-access program given in listings 5a and 5b. Since the file created by benchmark 5a is relatively large, it's doubtful that it could be stored on one large, contiguous chunk of disk. More than likely, it will be segmented into several pieces, depending upon how full the filesystem is. Most efficient UNIX (and UNIX-like) implementations segment a physical disk into more than one logical disk partition. Each partition is called a filesystem. When the filesystem is created, all disk blocks are contiguous. As the filesystem is used more and more, it becomes more splintered with many small chunks of contiguous space.

Since we would like to run the bench-

(continued)

Listing 2: *The system-call benchmark.*

```
/*
 *              UNIX Operating System Implementation Test #2
 *
 *  This program compounds the kernel overhead involved in executing
 *  a system call. Making a system call involves a 'trap' to kernel
 *  or supervisor mode, performing the desired function, and returning.
 *  Context switching is, when it occurs, also overhead. The getpid()
 *  system call is used because all it does is look in an in-core table
 *  for the numeric process id.
 *
 *  Instructions:
 *      Compile by:                 cc -O -s -o scall scall.c
 *
 *              The -O option says to use the optimizer.
 *              The -s option says to strip the namelist from the
 *                  object file after linking.
 *              The -o option says to place the object file in the file
 *                  specified by the next argument.
 *
 *      Time by:                    /bin/time scall
 *
 *  Results:
 *          Since we're testing system overhead, the elapsed time is of
 *          interest here.
 *
 */

#define TIMES  25000

main()
{
                /* take advantage of the hardware */
        register int i;
        for (i = 0; i < TIMES; i++)
                getpid();
}
```

marks under normal operating conditions, benchmark 5 should be executed in a filesystem that is used regularly. Several UNIX implementations place the directory /tmp in a filesystem of its own, since /tmp is used frequently under normal conditions. In any case, this benchmark should be run in an active filesystem in order to give a more realistic result as to what the response time under a real user load would be. This benchmark, of course, is extremely disk dependent, but that's what we're testing. As implied, the elapsed time is important here because the time spent waiting for I/O completion is not charged to either user or system time.

One of the things programmers do best is compile programs, and the compiler is a good operating-system exerciser because of it. The command to compile a C program under UNIX is **cc**. This command is actually a small C program that invokes the C preprocessor, the compiler proper, the assembler, and the linker in succession. To time the compilation process, just place **/bin/time** in front of the **cc** command line. Naturally, the C compiler is disk intensive, and with today's fast microprocessors, the disk is often the bottleneck in compilation throughput.

Something needs to be said about the size of the object files that the compiler leaves us with. It can be found by direct examination that the size of the object files compiled on comparable microcomputers can vary by an order of magnitude. In early UNIX days, when memory address space was limited, the loader didn't include a lot of unused code in the object file when it resolved all function references. With today's microcomputers having more memory than minicomputers of a few years ago, some implementations include unnecessary system-call hooks that are never referenced in the program. A good way to test this is to compile the following program:

```
main ()
{
}
```

which is the shortest C program possible. To tell how much memory the object file will use when loaded into memory, look at the size of the object file with the UNIX **size** command. **Size** reports the size of the text, data, and bss segments. The text segment is composed of program instructions. The data segment contains initialized program data. The bss segment contains uninitialized program data. The total size is usually given in both decimal and/or octal or hexadecimal. Another command of interest is **nm**, which will list the symbol table (NaMelist) of an object file. Some of the library modules loaded will be present in any program, and with good reason (**__exit**, **__environ**, **__cleanup**, **__main**, and **crt0.o**, for example). Some are pure excess (**malloc.o**, **isatty.o**, **write.o**, and **stty.o**, for example) and usually result from one library func-

Listing 3: *The user function-call benchmark.*

```
/*
 *              UNIX Operating System Implementation Test #3
 *
 * This program enables precise arithmetic calculations of user function
 * overhead by subtracting the execution user time when compiled without
 * using a function from execution user time using a function.
 *
 * Instructions:
 *    Compile by:          cc -O -DEMPTY -s -o fcalle fcall.c
 *                 and
 *                         cc -O -DASSIGN -s -o fcalla fcall.c
 *
 *              The -O option says to use the optimizer.
 *              The -D option specifies C preprocessor action.
 *              The -s option says to strip the namelist from the
 *                 object file after linking.
 *              The -o option says to place the object file in the file
 *                 specified by the next argument.
 *
 *    Time by:              /bin/time fcalle
 *              and
 *                         /bin/time fcalla
 *
 *    Results:
 *              Since the user time is more accurate than the real time,
 *              and since system time effectively does not contribute to
 *              the real time number, we can use the difference between
 *              the user times in seconds as an accurate numerical account
 *              of function call overhead.
 *
 */
#define TIMES  50000

main()
        /* The first way of doing things — use a function call */
#ifdef EMPTY
{
        register unsigned int i, j;
        for (i=0; i < TIMES; i++)
                j = empty(i);
}

        /* the empty function */
empty(k)
register unsigned int k;
{
        return(k);
}
#endif
#ifdef ASSIGN
        /*  The second way of doing things — without a function call */
{
        register unsigned int i, j;
        for (i = 0; i < TIMES; i++)
                j = i;
}
#endif
```

tion referencing another in a larger module, creating a cascade effect. The compactness of the code generated says something about the efficiency and implementation of the compiler and loader.

We've covered most of the more frequently used aspects of UNIX individually up to now. Let's develop some tests for the UNIX system interface, the shell. The best way to test this is by having a shell program do what users normally do when they sit down at the keyboard.

A good general UNIX benchmark is the shell script, or program, in listing 6a. This program, named **tst.sh**, invokes several commonly used UNIX commands and exercises disk access with them. This program was originally written for use in evaluating UNIX microcomputers at the '83 USENIX (an association of UNIX users) conference. In retrospect, it should have contained some commands to run concurrently in the background, such as the compilation of one of the C benchmarks described above. This benchmark makes use of the shell's I/O redirection and indirection (indirection being the ability to take input from the current input stream instead of a file) to sort, save on disk, manipulate, and ultimately remove from disk a list of English words. The utilities used (**sort**, which sorts; **od**, which gives an octal listing; **grep**, which does pattern matching; **tee**, which makes a disk copy of the input given it; **wc**, which counts words, lines, and characters; and **rm**, which removes disk files) are all standard UNIX tools. The shell variable **$$** is the current numerical process ID and is used to make unique filenames. The shell benchmark is run with the command **/bin/time /bin/sh tst.sh**. Execution times for even this simple benchmark varied widely, as shown in table 1.

A few words should be said about determining how many users a small multiuser system can support. With small multiuser systems, accurately simulating real user load is more important than with large multiuser systems because of the limited amount of memory, disk, and processor resources. You can simulate a real user load in several ways, but the only true way is to have someone at another terminal executing the same program you are at the same time. Why can't a process running in the background simulate a real user load? Because background processes usually run with a lower priority. Additionally, some multiuser microcomputer implementations limit the amount of memory an individual user can use at one time, even if no other user is on the system! What's more, some implementations impose an incredibly small limit on the number of files you can have open or the number of processes you can have running at any one time, again regardless of the number of other users or processes on the system. Watch out for these systems.

Since we're mainly concerned with microcomputer implementations, where there may or may not be additional terminals, and since we want portable benchmarks that can be run on any

(continued)

Listing 4: *The Sieve of Eratosthenes benchmark.*

```
/*
 *                  UNIX Operating System Implementation Test #4
 *
 * No benchmark suite would be complete without the ever-popular
 * sieve benchmark. It is a good test of compiler efficiency and
 * CPU throughput. Below is a sieve benchmark as presented in the
 * January 1983 issue of BYTE, with some minor changes: Register
 * declarations have been added, and some unnecessary (from our
 * standpoint) printf() statements removed.
 *
 * Instructions:
 *     Compile by:                  cc -O -s -o sieve sieve.c
 *
 *                 The -O option says to use the optimizer.
 *                 The -s option says to strip the namelist from the
 *                   object file after linking.
 *                 The -o option says to place the object file in the file
 *                   specified by the next argument.
 *
 *     Time by:                     /bin/time sieve
 *
 *     Results:
 *         In the past, the elapsed time has been used, since most
 *         operating systems can measure real time. Actually, user
 *         time is a better value.
 *
 */

/* Eratosthenes Sieve Prime Number program in C */
#define TRUE    1
#define FALSE   0
#define SIZE    8190

char flags[SIZE + 1];

main() {
    register int i, prime, k, count, iter;
/*  printf("10 iterations\n"); */                  /* We don't need this */
    for (iter = 1; iter <= 10; iter++) {           /* do program 10 times */
        count = 0;                                 /* prime counter */
        for (i = 0; i <= SIZE; i++)                /* set all flags TRUE */
            flags[i] = TRUE;
        for (i = 0; i <= SIZE; i++) {
            if (flags[i]) {                        /* found a prime */
                prime = i + i + 3;                 /* twice index + 3 */
/*              printf("\n%d", prime); */         /* Nor this */
                for (k = i + prime; k <= SIZE; k += prime)
                    flags[k] = FALSE;              /* kill all multiples */
                count++;                           /* primes found */
            }
        }
    }
/*  printf("\n%d primes.", count);*/               /* primes found on 10th pass */
}
```

UNIX system no matter how small, our only recourse is to benchmark a varying number of background processes (i.e., a multitasking benchmark) and assume that the results can be extrapolated to a multiuser environment. Even if the benchmark is used to help decide which single-user system to buy, evaluating background-process performance is beneficial since the ability to have many background processes is a strong point of UNIX.

Using the shell benchmark in listing 6a as a starting point, we can invoke that script in the background a number of times to see how long it takes to execute one, two, three, four, five, and even six of these identical background processes. The shell script in listing 6b does just that. Contained in a file called **multi.sh**, it executes the shell test found in listing 6a in the background a number of times. The number of background processes created is determined by the number of command-line parameters given the shell script. The actual values of the command-line parameters are not important, it's the quantity of positional parameters that the shell script uses. Although any character would do as a positional parameter, for readability it is convenient to use the characters "1," "2," "3," etc. as those parameters. The benchmark is run as shown in table 2.

The shell statement **wait** causes the shell script to pause until all background processes have terminated. Invoking **tst.sh** more than six times may not be possible (depending upon your operating system) if a "per-user process limit" is defined.

Table 3 shows the results from the multitasking shell benchmark given in listing 6b for a variety of UNIX-based systems. The table is sorted on the fastest elapsed time for six background processes. Remember, this benchmark does not measure how many users the system will support but is rather a measure of how many processes the system will support comfortably.

By plotting the number of invocations versus execution time, you can graph how a multitasking load varies with response time. See figure 1 for a plot of the results of table 3 in this manner. With fast disks the graph should be linear, with a change in slope when there are more processes than can remain concurrently in memory.

Listing 5a: A *benchmark to create and write a disk file.*

```
/*
 *              UNIX Operating System Implementation Test #5a
 *
 * This portion of the disk throughput benchmark creates and writes
 * a 512×256 byte file. Since UNIX is so disk intensive, it is important
 * to have some general idea of how fast (or slow) disk operations are.
 *
 * Instructions:
 *    Compile by:              cc -O -s -o dwrite dwrite.c
 *
 *             The -O option says to use the optimizer.
 *             The -s option says to strip the namelist from the
 *                object file after linking.
 *             The -o option says to place the object file in the file
 *                specified by the next argument.
 *
 *    Time By:                  /bin/time dwrite
 *
 *    Results:
 *       The time to observe is the elapsed time, as we are trying to
 *       gauge disk throughput.
 *
 */
#include <stdio.h>

#define BLOCKS  256

main()
{
        /* the buffer for writing */
    char buffer[512];
        /* the filename */
    char *filename = "a_large_file";
        /* a counter to keep up with the blocks written */
    register int i;
        /* file descriptor for the disk file */
    int fildes;
        /* create the file */
    if ((fildes = creat(filename, 0640)) < 0) {
        printf("Cannot create file\n");
        exit(1);
    } else {
        close(fildes);
                /* open the file for writing */
        if ((fildes = open(filename, 1)) < 0) {
            printf("Cannot open file\n");
            exit(1);
        }
    }
    for (i = 0; i < BLOCKS; i++)
                /* write the file, one block at a time */
        if (write(fildes, buffer, 512) < 0) {
            printf("Error writing block %d\n", i);
            exit(1);
        }
        /* close the file now that we're done */
    close(fildes);
}
```

Listing 5b: A *benchmark to randomly read the disk file created by listing 5a.*

```
/*
 *              UNIX Operating System Implementation Test #5b
 *
 * This portion of the benchmark opens and reads a 256×512 byte
 * file. This benchmark uses a random instead of sequential access
 * read, since the majority of disk access is random. Due to differences
```

(continued)

```
 *  in the rand() routine between UNIX versions, you need to determine if
 *  the rand() on the machine to be tested generates numbers in the range
 *  0 — 2^15 or in the range 0 — 2^31, and compile the benchmark accordingly.
 *
 *  Instructions:
 *      Compile By:         cc -DSIXTEEN -O -s -o dread dread.c
 *                  for machines with rand() in the range 0 — 2^15
 *
 *                          cc -DTHIRTYTWO -O -s -o dread dread.c
 *                  for machines with rand() in the range 0 — 2^31
 *
 *                  The -O option says to use the optimizer.
 *                  The -s option says to strip the namelist from the
 *                      object file after linking.
 *                  The -o option says to place the object file in the file
 *                      specified by the next argument.
 *
 *      Time By:            /bin/time dread
 *
 *      Results:
 *          The time to observe is the elapsed time, as we are trying to
 *          gauge disk throughput.
 *
 */

#include <stdio.h>
#define BLOCKS  256

long lseek();

main()
{
                /* the buffer for writing */
        char buffer[512];
                /* the filename */
        char *filename = "a_large_file";
                /* a counter counting blocks read */
        register int i;
                /* the file descriptor */
        int fildes;
                /* offset to seek into file */
        long int offset;

                /* open the file */
        if ((fildes = open(filename, 0)) < 0) {
                printf("Cannot find '%s'. Run 'dwrite' first.\n", filename);
                exit(1);
        }

        for (i = 0; i < BLOCKS; i++) {
                        /* pick a byte, any byte */
#ifdef SIXTEEN
                offset = (long)rand() * 4L;
#endif
#ifdef THIRTYTWO
                offset = (long)rand() / 16384L;
#endif
                        /* seek to it */
                if (lseek(fildes, offset, 0) < 0L) {
                    printf("Lseek to %ld failed i=%d\n", offset, i);
                    exit(1);
                }
                        /* read a block, starting with the current byte */
                if (read(fildes, buffer, 512) < 0) {
                    printf("Error reading block at byte %ld\n",offset);
                    exit(1);
                }
        }
                /* get rid of the file */
        unlink(filename);
}
```

A short benchmark that tests incrementing and looping is shown in listing 7. It originally appeared on UNIX USENET news (article megatest.186) in February 1983. This little benchmark tests long integer arithmetic (increment and test) and is totally processor bound. It is a lot like the functional benchmarks shown earlier; it tests long integer arithmetic but does little else. It could be improved by multiplying by 2, dividing by 2, adding 2, and then subtracting 1 to better test long integer arithmetic functions. The benchmark is presented here in its original form because I had already tested a number of machines with that particular version. See the results in table 1.

Results

A lot has happened in the last nine months during which this article was written. Several UNIX implementations now exist for the IBM PC. Microcomputer UNIX systems continue to infiltrate the business environment, and the UNIX application-software market seems to be developing at a good pace. Both DEC and IBM have embraced UNIX as an alternative to their own proprietary operating systems, which lends legitimacy to the claim that UNIX is an industry-standard operating system. There does not yet seem to be a clear winner in the UNIX microcomputer marketplace though several vendors are in the forefront of the cost/performance ratio contest.

Judging from the systems I've seen, the best performance comes from the Altos 586. It has less memory and fewer I/O ports than the Altos 986 but is otherwise identical. For about $10,000, you get an excellent multiuser UNIX system (512K-byte RAM, 40-megabyte [formatted] Winchester, and six serial ports) that under moderate load approaches DEC VAX performance for most tasks that a user would normally invoke. Some may argue that if the operating system isn't spelled U-N-I-X, it isn't real UNIX. That's just not the case. Altos XENIX *is* Version 7 UNIX with some useful extensions, including a screen-oriented editor, record and file locking, and semaphores. Although AT&T no longer markets Version 7 UNIX, it is well established in the marketplace and will be around for quite a while.

(continued)

The Sun-2/120 and Masscomp computers *are* VAX-class machines, but their cost is beyond the reach of most prospective microcomputer owners. They both offer superb graphics and excellent response time under loading.

The TRS-80 16B is a usable multiuser microcomputer system, but its response time is hindered by the relatively slow internal 15-megabyte Winchester. Thus, depending upon the applications run, it may not be desirable for more than a two-user load.

The SCI-1000 system benchmarked is still under development, and the times reported here should not be taken as gospel. This system, with an 80186 chip, has the potential for better performance than the Altos 586 at less cost and offers System III UNIX.

IBM's UNIX, PC/IX, was not developed in house. It is a System III port with added features (a vi-like full-screen editor) done by Interactive Systems Corporation. It is a complete, usable single-user implementation that does what can be done with the 8088. It's interesting to note that both IBM PC (16-/8-bit 8088) implementations performed better than did the old reliable 16-bit F-11 chip used in the PDP-11/23 and DEC Professional.

The Omnibyte OB68K with Idris was one of the first UNIX work-alike systems around. As such, the implementation is not the de facto Version 7 standard. I understand that a new version of Idris is coming out (to borrow a phrase from Jerry Pournelle) Real Soon Now for the Omnibyte that increases performance substantially.

The VENIX implementations on the DEC Professional and IBM PC perform adequately but seem to have a problem with multiple background processes. Although it makes sense to limit the number of processes a user may have on a multiuser minicomputer, it doesn't make much sense to impose those

Listing 6a: A *general-purpose shell benchmark. The shell script shown is contained in a file called* tst.sh *and is invoked by* /bin/time /bin/sh tst.sh.

```
sort >sort.$$ <<'EOF'
Now
is
the
time
for
all
good
men
to
come
to
the
aid
of
their
country
'EOF'
od sort.$$ | sort -n +1 > od.$$
grep the sort.$$ | tee grep.$$ | wc > wc.$$
rm sort.$$ grep.$$ od.$$ wc.$$
```

Listing 6b: A *multitasking benchmark with a variable number of background processes. This shell script is contained in a file called* multi.sh. *The number of concurrent processes created is determined by the number of command-line parameters, such as* /bin/time /bin/sh multi.sh 1 2 3.

```
for i
do
        echo $i
        /bin/sh tst.sh &
done
wait
```

limits on a microcomputer that will probably never be used by more than one person. What's more, no message is given the user when the number of processes reaches the per-user process limit. Instead, quite literally, nothing happens. Granted that the multitasking benchmark is a little esoteric, it is the only way to simulate a multiuser/multitasking load short of having multiple users and is a good measure of how efficiently or inefficiently competing background processes are handled.

And then there's Apple's Lisa. Due to disk I/O limitations, Lisa's in a class by herself when it comes to disk-intensive tasks, as can be easily seen from figure 1. This exemplifies my claim that disk I/O is the single most limiting factor in overall response time and system throughput. If Apple could improve the disk throughput for Lisa to the same as an Altos, Lisa would rival the Altos in the best-value category, not to mention the possibility of excellent graphics.

It should be noted that some of the systems above that were implied to be single-user are really multiuser, but the response time is such that they would not be usable in a multiuser environment. This is the case with such computers as the IBM PC, DEC Professional, and Apple Lisa.

CONCLUSIONS

Some words of caution: a few microcomputer systems that claim to be

(continued)

Table 2: *The shell benchmark run sequence.*

```
/bin/time /bin/sh multi.sh 1
/bin/time /bin/sh multi.sh 1 2
/bin/time /bin/sh multi.sh 1 2 3
/bin/time /bin/sh multi.sh 1 2 3 4
/bin/time /bin/sh multi.sh 1 2 3 4 5
/bin/time /bin/sh multi.sh 1 2 3 4 5 6
```

Table 3: *Results for the multitasking* UNIX *benchmark in listing 6b with a variable number of background processes. The data are the elapsed (real) times for the benchmark to complete. The table is sorted on the fastest execution times with six background processes (the last column) where possible.*

System			Elapsed (Real) Time in Seconds					
			Number of Concurrent Processes					
No.	Machine	UNIX Version	1	2	3	4	5	6
1	VAX-11/780	4.1 BSD	4.3	5.5	7.8	9.0	11.0	13.8
2	VAX-11/750	4.1 BSD	4.3	5.5	8.8	10.3	13.3	15.0
3	PDP-11/70	2.8 BSD	5.0	7.8	9.3	11.8	14.3	16.7
4	Masscomp	Sys III+	4.2	5.5	9.1	11.8	14.5	17.8
5	Sun-2/120	4.2 BSD	3.6	6.2	8.7	11.8	14.4	18.0
6	Altos 986	XENIX	6.3	7.3	9.3	19.3	27.2	36.0
7	TRS-80 16B	XENIX	20.0	24.5	33.0	56.5	1:10.5	1:39.3
8	SCI-1000	Sys III+	15.1	28.6	51.8	1:17.4	1:34.8	1:57.2
9	PDP-11/23	V7	22.3	37.3	52.3	1:14.8	1:31.0	2:05.0
10	IBM PC XT	PC/IX	10.6	23.4	42.8	1:14.1	1:24.2	2:10.7
11	Apple Lisa	Sys III+	38.1	1:14.8	1:54.5	2:34.2	3:14.6	3:48.6
12	PDP-11/23	VENIX	14.0	32.8	—	—	—	—
13	IBM PC XT	VENIX/86	15.0	23.5	39.0	—	—	—
14	DEC Pro/350	VENIX	26.0	41.0	1:22.3	—	—	—
15	Omnibyte	Idris 1.2(*)	—	—	—	—	—	—

\+ Indicates UNIX System III plus some Berkeley enhancements.
– Indicates a benchmark that would not complete.
* The Idris shell command **wait** did not appear to function properly, and thus the benchmark could not be run.

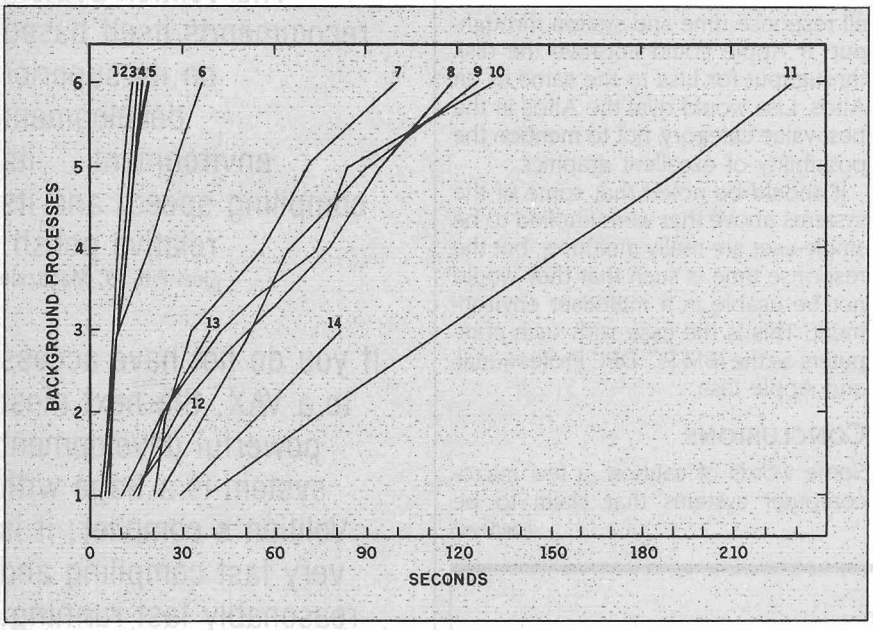


Figure 1: A *graph of the multitasking benchmark data in table 2 with the number of background processes versus elapsed time for each computer. The numbers at the top of each line correspond to the computers as listed in table 3. It is interesting to note the cluster of high-performance systems on the left-hand side and the cluster of other systems on the right.*

Listing 7: A *simple benchmark to test incrementing and looping.*

```
/*
*       UNIX Operating System Implementation Test #7
*
* This program tests long integer incrementation. It is
* taken from USENET news article "megatest.186".
*
* Instructions:
*    Compile by:                cc -O -s -o loop loop.c
*
*            The -O option says to use the optimizer.
*            The -s option says to strip the namelist from the
*               object file after linking.
*            The -o option says to place the object file in the file
*               specified by the next argument.
*
*    Time by:                   /bin/time loop
*
*    Results:
*         Although not very significant, it does say something about the
*         speed of the processor, since the compiler would hopefully
*         compile the "i++" as an INCR instruction and not an ADD
*         instruction. The benchmark is presented here for historical
*         reasons.
*
*/
main()
{
        long i;
        for (i = 0; i < 1000000; i++)
                ;
        printf("Done\n");
}
```

multiuser and UNIX-like do not swap. That is, they cannot swap a process out to disk and bring in another user's process. A system that cannot swap is neither truly multiuser nor UNIX-like. When a process runs out of primary memory in these systems, it dies. These implementations are substantially cheaper than most others, so be suspicious of low-cost UNIX-like systems.

As mentioned earlier, some systems implement a relatively low predefined limit on the amount of memory or number of processes one user can have, regardless of other system activity (or inactivity). Once this limit is exceeded, activity grinds to a halt, as a deadlock has been reached. Each blocked process (blocked in the sense that it is waiting for resources before it can continue) is waiting for the other to terminate before it can continue. If you plan to be an active user on a small multiuser system in a single-user environment, look out for this. The multitasking benchmark in listing 6b will usually bring any problems to light.

Knowledgeable 4.1 BSD and 4.2 BSD users should beware of systems that claim to have Berkeley enhancements. This means that the Berkeley version of some UNIX commands have been added. For example, most so-called Berkeley-enhanced systems include the **termcap** (terminal capability) database, **more** (a utility that prints files one screen at a time), and a version of **ls** (a utility to list the files in a directory) that lists files across, rather than down the screen. Don't expect to find the **newtty** driver and the job-control facilities of real Berkeley UNIX systems.

If you're considering a UNIX microcomputer, remember that response doesn't always vary linearly with load (even on large UNIX systems). This is due to several factors, most notably available real memory and disk-access speed. If you plan to add a user or two later, test the prospective system now. Find out if the hardware can support additional memory and/or faster disks.

The benchmarks presented here try not to be blind to what users do at the keyboard (not all users execute programs similar to the Sieve of Eratosthenes), but they do try to evaluate operating-system features that are routinely used. By explaining how benchmarks should be developed, this article

Benchmark the specific kinds of things you will be doing as well as overall performance.

has tried to dispel the myth that all benchmarks do is see how fast a machine can crunch numbers (e.g., the Whetstone benchmark has not been mentioned).

Of course, benchmark results are not the only means to judge microcomputers. Clear and sufficient documentation, a solid customer base, and good product-support history are also important. If you do perform benchmarks on systems you are considering purchasing, try to benchmark the specific kinds of things you will be doing as well as overall performance in case your needs change. This sounds incredibly obvious, but many people have been disappointed by systems purchased yesterday that don't meet their needs today. ■

REFERENCES

1. Gilbreath, Jim, and Gary Gilbreath. "Eratosthenes Revisited: Once More through the Sieve." BYTE, January 1983, page 283.
2. Lions, J. "A Commentary on the UNIX Operating System." 1977.
3. Peterson, J., and A. Silberschatz. *Operating System Concepts*. Reading, MA: Addison-Wesley, 1983.
4. UNIX *Programmer's Manual*, 7th ed., Virtual VAX-11 Version, volumes 1 and 2c. Berkeley, CA: University of California, Department of Electrical Engineering and Computer Science, June 1981.
5. UNIX *Programmer's Manual*, 7th ed., volumes 1, 2a, and 2b. Murray Hill, NJ: Bell Telephone Laboratories, January 1979.

ACKNOWLEDGMENTS

Thanks to Ellen Mendelson and Walt Kennedy of the Durham and Raleigh Radio Shacks for access to the TRS-80 Model 16. Thanks to David Holloman of Keystone Systems Inc. of Raleigh for access to the Altos 586. Thanks to Michael Smith of East Carolina University for running the Idris benchmarks. Special thanks to the North Carolina Educational Computing Service for allowing me access to most of the remaining machines listed in table 1.