A UNIX CLONE WITH SOURCE CODE FOR OPERATING SYSTEMS COURSES

Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Postbus 7161
1007 MC Amsterdam, The Netherlands
USENET: ast@cs.vu.nl

1. INTRODUCTION

Students learn by doing, not by listening. Physicists and chemists have long understood this, which is why students in these fields are required to perform experiments in the laboratory and write up their findings. Computer scientists also realize this basic truth, so many courses require the students to do some programming. For courses like *Introduction to Pascal* and *Data Structures 101* it is not hard to dream up suitable exercises. For courses on operating systems, it is much more difficult to arrange for a suitable programming laboratory. This paper discusses a solution to the problem based on using a UNIX clone written by the author and available in source form for classroom use.

The essential problem with operating systems is that even the small and simple ones are large and complex. The large and complex ones are unspeakable. It is simply not feasible to ask the students to write an operating system, no matter how simple, as an exercise for a one semester course. No one would pass.

At the other end of the spectrum, the students can be asked to write a tiny piece of an operating system, such as a CPU scheduler, all by itself. This strategy has the advantage that everybody passes. Unfortunately, nobody learns anything about operating systems this way. The books and lecture material for most courses are already heavily skewed toward the few topics that can be pulled out of context and isolated, such as monitors, schedulers, and deadlock algorithms. Building the laboratory around these topics as well just makes things worse.

The only reasonable approach is to distribute to the students a working, well-documented operating system that they can study in detail and then modify. In this way they get to see a whole system close up, without having to write all the code. This strategy is clearly attractive but it has three basic problems:

- hardware
- software
- documentation

Let us deal with these one at a time.

Operating systems generally have to run on the bare machine. For a course in which the only computing facility available is a large time-shared computer (or, heaven forbid, a batch system), there are obvious problems. Fortunately, substantial microcomputers have now become so inexpensive that acquiring 20 machines just for the operating systems lab is becoming a real possibility at many schools. There are IBM PC and even

AT clones available for under $1000 these days, which makes them only slightly more expensive than ordinary terminals, and a lot more flexible.

The real catch is the software. About 10 years ago, many universities taught courses based on Version 6 of the UNIX† operating system, using a little booklet written by John Lions of the University of New South Wales as the text. Lions' commentary fell into disuse when AT&T changed the licensing agreement starting with Version 7 to prohibit using the UNIX code in courses.

Faced with this problem for my own students, and seeing no solution in sight, several years ago I simply bit the bullet and sat down and wrote a new operating system for classroom use from scratch. This system, now finished, is called MINIX and is functionally compatible with Version 7 of the UNIX system (i.e., has the same system calls). Internally it is entirely new with a completely different (and much more modular) structure than UNIX. The file system, for example, runs outside the kernel as a message driven file server. Best of all, neither the operating system kernel, nor any of the utility programs (*cat*, *cp*, *grep*, *ls*, *make*, etc.) contain even a single line of AT&T code. Consequently, this system can be distributed to students for course use.

MINIX runs on the IBM PC, XT, and AT, and on those clones that are 100% hardware compatible. It does not use the IBM BIOS because the BIOS does all I/O, even terminal input, by busy waiting rather than using interrupts. Therefore MINIX contains I/O drivers that directly access the PC's peripheral chips. If a clone has different peripheral chips, MINIX will not run. Empirically, it seems to work on about 3/4 of the clones tested.

Version 7 was chosen as the base because the goal of this exercise was to make a system that students could understand. In contrast, x.y (x = [4, 5]; y = [1, 2, 3]) is hairy enough to drive a wizard to distraction, let alone an unsuspecting undergraduate. Besides, MINIX was designed to run on a PC with or without a hard disk, to keep the necessary hardware cheap. Putting x.y onto a 256K IBM PC with one 360K floppy disk and no hard disk is left as an exercise for the reader.

The last element needed to make this system suitable for classroom use is documentation. To deal with this issue, I jotted down some notes. These have now been published as a 719 page book (Tanenbaum, 1987). The book covers both the usual "theory" of operating systems (interprocess communication, scheduling, memory management, etc.) and then shows how these topics are actually applied in MINIX. Of the 719 pages, 253 are the kernel listing, in C. The rest is in English (sort of).

To make MINIX useful in those situations where PC's are not available, I have also written a complete IBM PC simulator in C, so that MINIX can be run on the simulator on top of UNIX on an ordinary time-sharing machine. Unfortunately, the simulator (which is basically just an 8088 interpreter) consumes a lot of CPU time and is correspondingly slow. Therefore I also wrote a package that allows the MINIX file system to be compiled and run on a VAX at normal speed. While having the students only be able to experiment with the file system is not as good as giving them the whole system, it is better than nothing at all.

UNIX is a registered trademark of AT&T Bell Laboratories.

2. THE USER VIEW OF MINIX

In this section we will briefly sketch what MINIX looks like to the user. In subsequent sections, we will take a closer look at its internal design. To run MINIX, you put the boot diskette into the IBM PC, XT, or AT, and turn the machine on. When the operating system has been loaded into memory, it asks the user to insert the root file system diskette, which is then copied to the RAM disk. After the RAM disk has been loaded, the system comes up and executes the shell script */etc/rc*, which typically contains commands to mount hard or floppy disk file systems, and so on. When */etc/rc* is finished, a "login:" message appears on the terminal. To the naive user, from this point on, the system is virtually indistinguishable from Version 7 of the UNIX system. Sophisticated users will notice that some V7 programs (e.g., *bc*, *m4*, *ptx*, *spline*, and *yacc*) are missing. Among the programs that are present (and compatible with their V7 counterparts) are:

ar basename cat cc chmod chown cmp comm cp date dd df echo grep head kill ln login lpr ls make mkdir mkfs mknod mount mv od passwd pr pwd rev rm rmdir roff sh size sleep sort split stty su sum sync tail tar tee time touch tr true umount uniq update wc

There is also a full Kernighan & Ritchie (1978) compatible C compiler derived from the Amsterdam Compiler Kit (Tanenbaum, 1983), a version of *make*, a shell compatible with the standard Bourne shell, a full-screen editor loosely inspired by *emacs*, programs to read and write MS-DOS diskettes, and a variety of programs not present in V7.

The situation at the system call level is similar. All the V7 system calls are available, except for a few of the more obscure ones. The following calls are fully implemented:

access alarm brk chdir chmod chown chroot close creat dup exec exit fork fstat getgid getpid getuid ioctl kill link lseek mknod mount open pause pipe read setgid setuid signal stat stime sync time times umask umount unlink utime wait write

Note that *fork* and *exec* are present, so MINIX, is a full multiprogramming system. The command

```
cc file.c &
```

runs the compilation in the background, just as you would expect.

In addition to the above utility programs and system calls, MINIX also has over 100 library procedures. These include the system call library (e.g., *pipe* and *read*), standard I/O (e.g., *fopen* and *fprintf*), string handling (e.g, *strcmp* and *strcpy*), and various miscellaneous procedures (e.g., *atoi* and *malloc*). Like the kernel and the utilities, these, too, have all been written from scratch without using the AT&T code as a guide or base.

3. OVERVIEW OF THE MINIX SYSTEM ARCHITECTURE

In this and the following sections, we will discuss the architecture of the MINIX system in some detail. Unlike UNIX which is organized as a single monolithic program that is loaded into memory at system boot time and then run, the MINIX kernel itself is structured in a much more modular way, as a collection of processes that communicate with each other and with user processes by sending and receiving messages. There are

separate processes for the memory manager, the file system, for each device driver, and for certain other system functions. This structure enforces a better interface between the pieces. The file system cannot, for example, accidentally change the memory manager's tables because the file system and memory manager each have their own address space.

These system processes are each full-fledged processes, with their own memory allocation, process table entry and state. They can be run, blocked, and send messages, just as the user processes. In fact, the memory manager and file system each run in user space as ordinary processes. The device drivers are all linked together with the kernel into the same binary program, but they communicate with each other and with the other processes by message passing.

When the system is compiled, four binary programs are independently created: the kernel (including the driver processes), the memory manager, the file system, and *init* (which reads */etc/ttys* and forks off the login processes). In other words, compiling the system results in four distinct *a.out* files. When the system is booted, all four of these are read into memory from the boot diskette.

It is possible, and in fact, normal, to modify, recompile, and relink, say, the file system, without having to relink the other three pieces. This design provides a high degree of modularity by dividing the system up into independent pieces, each with a well-defined function and interface to the other pieces. The pieces communicate by sending and receiving messages.

The various processes are structured in four layers:

4. The user processes (top layer).
3. The server processes (memory manager and file system).
2. The device drivers, one process per device.
1. Process and message handling (bottom layer).

Let us now briefly summarize the function of each layer.

Layer 1 is concerned with doing process management including CPU scheduling and interprocess communication. When a process does a SEND or RECEIVE, it traps to the kernel, which then tries to execute the command. If the command cannot be executed (e.g., a process does a RECEIVE and there are no messages waiting for it), the caller is blocked until the command can be executed, at which time the process is reactivated. When an interrupt occurs, layer 1 converts it into a message to the appropriate device driver, which will normally be blocked waiting for it. The decision about which process to run when is also made in layer 1. A priority algorithm is used, giving device drivers higher priority over ordinary user processes, for example.

Layer 2 contains the device drivers, one process per major device. These processes are part of the kernel's address space because they must run in kernel mode to access I/O device registers and execute I/O instructions. Although the IBM PC does not have user mode/kernel mode, most other machines do, so this decision has been made with an eye toward the future. To distinguish the processes within the kernel from those in user space, the kernel processes are called **tasks**.

Layer 3 contains only two processes, the memory manager and the file system. They are both structured as **servers**, with the user processes as **clients**. When a user

process (i.e., a client) wants to execute a system call, it calls, for example, the library procedure *read* with the file descriptor, buffer, and count. The library procedure builds a message containing the system call number and the parameters and sends it to the file system. The client then blocks waiting for a reply. When the file system receives the message, it carries it out and sends back a reply containing the number of bytes read or the error code. The library procedure gets the reply and returns the result to the caller in the usual way. The user is completely unaware of what is going on here, making it easy to replace the local file system with a remote one.

Layer 4 contains the user programs. When the system comes up, *init* forks off login processes, which then wait for input. On a successful login, the shell is executed. Processes can fork, resulting in a tree of processes, with *init* at the root. When CTRL-D is typed to a shell, it exits, and *init* replaces the shell with another login process.

4. LAYER 1 - PROCESSES AND MESSAGES

The two basic concepts on which MINIX is built are processes and messages. A process is an independently schedulable entity with its own process table entry. A message is a structure containing the sender's process number, a message type field, and a variable part (a union) containing the parameters or reply codes of the message. Message size is fixed, depending on how big the union happens to be on the machine in question. On the IBM PC it is 24 bytes.

Three kernel calls are provided:

- RECEIVE(source, &message)
- SEND(destination, &message)
- SENDREC(process, &message)

These are the only true system calls (i.e., traps to the kernel). RECEIVE announces the willingness of the caller to accept a message from a specified process, or ANY, if the RECEIVER will accept any message. (From here on, "process" also includes the tasks.) If no message is available, the receiving process is blocked. SEND attempts to transmit a message to the destination process. If the destination process is currently blocked trying to receive from the sender, the kernel copies the message from the sender's buffer to the receiver's buffer, and then marks them both as runnable. If the receiver is not waiting for a message from the sender, the sender is blocked.

The SENDREC primitive combines the functions of the other two. It sends a message to the indicated process, and then blocks until a reply has been received. The reply overwrites the original message. User processes use SENDREC to execute system calls by sending messages to the servers and then blocking until the reply arrives.

There are two ways to enter the kernel. One way is by the trap resulting from a process' attempt to send or receive a message. The other way is by an interrupt. When an interrupt occurs, the registers and machine state of the currently running process are saved in its process table entry. Then a general interrupt handler is called with the interrupt number as parameter. This procedure builds a message of type INTERRUPT, copies it to the buffer of the waiting task, marks that task as runnable (unblocked), and then calls the scheduler to see who to run next.

The scheduler maintains three queues, corresponding to layers 2, 3, and 4, respectively. The driver queue has the highest priority, the server queue has middle priority, and the user queue has lowest priority. The scheduling algorithm is simple: find the highest priority queue that has at least one process on it, and run the first process on that queue. In this way, a clock interrupt will cause a process switch if the file system was running, but not if the disk driver was running. If the disk driver was running, the clock task will be put at the end of the highest priority queue, and run when its turn comes.

In addition to this rule, once every 100 msec, the clock task checks to see if the current process is a user process that has been running for at least 100 msec. If so, that user is removed from the front of the user queue and put on the back. In effect, compute bound user processes are run using a round robin scheduler. Once started, a user process runs until either it blocks trying to send or receive a message, or it has had 100 msec of CPU time. This algorithm is simple, fair, and easy to implement.

5. LAYER 2 - DEVICE DRIVERS

Like all versions of UNIX for the IBM PC, MINIX does not use the ROM BIOS for input or output because the BIOS does not support interrupts. Suppose a user forks off a compilation in the background and then calls the editor. If the editor tried to read from the terminal using the BIOS, the compilation (and any other background jobs such as printing) would be stopped dead in their tracks waiting for the the next character to be typed. Such behavior may be acceptable in the MS-DOS world, but it certainly is not in the UNIX world. As a result, MINIX contains a complete set of drivers that duplicate the functions of the BIOS. Like the rest of MINIX, these drivers are written in C, not assembly language.

Each device driver is a separate process in MINIX. At present, the drivers include the clock driver, terminal driver, various disk drivers (e.g., RAM disk, floppy disk, hard disk), and printer driver. Each driver has a main loop consisting of three actions:

1. Wait for an incoming message.
2. Perform the request contained in the message.
3. Send a reply message.

Request messages have a standard format, containing the opcode (e.g., READ, WRITE, or IOCTL), the minor device number, the position (e.g., disk block number), the buffer address, the byte count, and the number of the process on whose behalf the work is being done.

As an example of where device drivers fit in, consider what happens when a user wants to read from a file. The user sends a message to the file system. If the file system has the needed data in its buffer cache, they are copied back to the user. Otherwise, the file system sends a message to the disk task requesting that the block be read into a buffer within the file system's address space (in its cache). Users may not send messages to the tasks directly. Only the servers may do this.

MINIX supports a RAM disk. In fact, the RAM disk is always used to hold the root device. When the system is booted, after the operating system has been loaded, the user

is instructed to insert the root file system diskette. The file system then sees how big it is, allocates the necessary memory, and copies the diskette to the RAM disk. Other file systems can then be mounted on the root device.

This organization puts important directories such as */bin* and */tmp* on the fastest device, and also makes it easy to work with either floppy disks or hard disks or a mixture of the two by mounting them on */usr* or */user* or elsewhere. In any event, the root device is always in the same place.

In the standard distribution, the RAM disk is about 240K, most of which is full of parts of the C compiler. In the 256K system, a much smaller RAM disk has to be used, which explains why this version has no C compiler: there is no place to put it. (The */usr* diskette is completely full with the other utility programs and one of the design goals was to make the system run on a 256K PC with 1 floppy disk.) Users with an unusual configuration such as 256K and three hard disks are free to juggle things around as they see fit.

The terminal driver is compatible with the standard V7 terminal driver. It supports cooked mode, raw mode, and cbreak mode. It also supports several escape sequences, such as cursor positioning and reverse scrolling because the screen editor needs them.

The printer driver copies its input to the printer character for character without modification. It does not even convert line feed to carriage return + line feed. This makes it possible to send escape sequences to graphics printers without the driver messing things up. MINIX does not spool output because floppy disk systems rarely have enough spare disk space for the spooling directory. Instead one normally would print a file *f* by saying

```
lpr <f &
```

to do the printing in the background. The *lpr* program insert carriage returns, expands tabs, and so on, so it should only be used for straight ASCII files. On hard disk systems, a spooler would not be difficult to write.

6. LAYER 3 - SERVERS

Layer 3 contains two server processes: the memory manager and the file system. They are both structured in the same way as the device drivers, that is a main loop that accepts requests, performs them, and then replies. We will now look at each of these.

The memory manager's job is to handle those system calls that affect memory allocation, as well as a few others. These include FORK, EXEC, WAIT, KILL, and BRK. The memory model used by MINIX is exceptionally simple in order to accommodate computers without any memory management hardware. When the shell forks off a process, a copy of the shell is made in memory. When the child does an EXEC, the new core image is placed in memory. Thereafter it is never moved. MINIX does not swap or page.

The amount of memory allocated to the process is determined by a field in the header of the executable file. A program, *chmem*, has been provided to manipulate this field. When a process is started, the text segment is set at the very bottom of the allocated memory area, followed by the data and bss. The stack starts at the top of the

allocated memory and grows downward. The space between the bottom of the stack and the top of the data segment is available for both segments to grow into as needed. If the two segments meet, the process is killed.

In the past, before paging was invented, all memory allocation schemes worked like this. In the future, when even small microcomputers will use 32-bit CPUs and 1M x 1 bit memory chips, the minimum feasible memory will be 4 megabytes and this allocation scheme will probably become popular again due to its inherent simplicity. Thus the MINIX scheme can be regarded as either hopelessly outdated or amazingly futuristic, as you prefer.

The memory manager keeps track of memory using a list of holes. When new memory is needed, either for FORK or for EXEC, it searches the hole list and takes the first hole that is big enough (first fit). When a process terminates, if it is adjacent to a hole on either side, the process' memory and the hole are merged into a bigger hole.

The file system is really a remote file server that happens to be running on the user's machine. However it is straightforward to convert it into a true network file server. All that needs to be done is change the message interface and provide some way of authenticating requests. (In MINIX, the source field in the incoming message is trustworthy because it is filled in by the kernel.) When running remote, the MINIX file server maintains state information, like RFS and unlike NFS.

The MINIX file system is similar to that of V7 UNIX. The i-node is slightly different, containing only 9 disk addresses instead of 13, and only 1 time instead of 3. These changes reduce the i-node from 64 bytes to 32 bytes, to store more i-nodes per disk block and reduce the size of the in-core i-node table.

Free disk blocks and free inodes are kept track of using bit maps rather than free lists. The bit maps for the root device and all mounted file systems are kept in memory. When a file grows, the system makes a definite effort to allocate the new block as close as possible to the old ones, to minimize arm motion. Disk storage is not necessarily allocated one block at a time. A minor device can be configured to allocate 2, 4 (or more) contiguous blocks whenever a block is allocated. Although this wastes disk space, these multiblock **zones** improve disk performance by keeping file blocks close together. The standard parameters for MINIX as distributed are 1K blocks and 1K zones (i.e., just 1 block per zone).

MINIX maintains a buffer cache of recently used blocks. A hashing algorithm is used to look up blocks in the cache. When an i-node block, directory block, or other critical block is modified, it is written back to disk immediately. Data blocks are only written back at the next SYNC or when the buffer is needed for something else.

The MINIX directory system and format is identical to that of V7 UNIX. File names are strings of up to 14 characters, and directories can be arbitrarily long.

7. LAYER 4 - USER PROCESSES

This layer contains *init*, the shell, the editor, the compiler, the utilities, and all the user processes. These processes may only send messages to the memory manager and the file system, and these servers only accept valid system call requests. Thus the user processes do not perceive MINIX to be a general-purpose message passing system.

However, removing the one line of code that checks if the message destination is valid would convert it into a much more general system (but less UNIX-like).

8. MINIX DOCUMENTATION

For a system one of whose purposes is teaching about operating systems, ample documentation is essential. For this reason I have written an ample textbook (719 pages) treating both the theory and the practice of operating system design (Tanenbaum, 1987). The table of contents is as follows:

CHAPTERS

1. Introduction
2. Processes
3. Input/Output
4. Memory Management
5. File Systems
6. Bibliography and Suggested Readings

APPENDICES

A. Introduction to C
B. Introduction to the IBM PC
C. MINIX Users Guide
D. MINIX Implementers Guide
E. MINIX Source Code Listing
F. MINIX Cross Reference Map

The heart of the book is chapters 2-5. Each chapter deals with the indicated topic in the following way. First comes a thorough treatment of the relevant principles (thorough enough to be usable as a university textbook on operating systems). Next comes a general discussion of how the principles have been applied in MINIX. Finally there is a procedure by procedure description of how the relevant part of MINIX works in detail. The source code listing of appendix E contains line numbers, and these line numbers are used throughout the book to pinpoint the code under discussion. The source code itself contains more than 3000 comments, some more than a page long. Studying the principles and seeing how they are applied in a real system gives the student a better understanding of the subject than either the principles or the code alone would.

Appendices A and B are introductions to C and the IBM PC for readers not familiar with these subjects. Appendix C tells how to boot MINIX, how to use it, and how to shut it down. It also contains all the manual pages for the utility programs. Most important of all, it gives the super-user password.

Appendix D is for people who wish to modify and recompile MINIX. It contains a wealth of nutsy-boltsy information about everything from how to use MS-DOS as a development system, to what to do when your newly made system refuses to boot.

Appendix E is a full listing of the operating system. The utilities (mercifully) are not listed.

9. DISTRIBUTION OF THE SOFTWARE

The software distribution is being done by Prentice-Hall. The book contains a business reply card for ordering the software. Four packages are available. All four contain the full source code; they differ only in the configuration of the binary supplied. All cost $79.95. The four packages are:

- 640K IBM PC version
- 256K IBM PC (no C compiler)
- IBM PC-AT (512K minimum)
- Industry standard 9-track tape

The 640K version will also run on 512K systems, but it may be necessary to *chmem* parts of the C compiler to make it fit. The tape version also contains the IBM PC simulator and other software needed for classroom use on a VAX or other time sharing machine. The software packages do not include the book.

A few words about the legal status of the software are in order. The software has been copyright by Prentice-Hall. It is *not* public domain. However, Prentice-Hall is permitting copies of both the binaries and sources to be made for educational use without requiring a license or payment. Thus professors can legally make copies of the software for their students. For commercial use of the software (e.g., porting it to CPUs other than the Intel family and then reselling it), written permission from Prentice-Hall is required.

If there is sufficient interest, a USENET newsgroup comp.os.minix will be set up. This channel can be used by people wishing to contribute new programs, point out and correct bugs, discuss the problems of porting MINIX to new systems, etc.

10. ACKNOWLEDGEMENTS

I would like to thank the following people for contributing utility programs and advice to the MINIX effort: Martin Atkins, Erik Baalbergen, Charles Forsyth, Richard Gregg, Michiel Huisjes, Patrick van Kleef, Adri Koppes, Paul Ogilvie, Paul Polderman, and Robbert van Renesse. Without their help, the system would have been far less useful than it now is.

11. REFERENCES

Kernighan, B.W., and Ritchie, D.M.: *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

Tanenbaum, A.S., van Staveren, H., Keizer, E.G., and Stevenson, J.W.: "A Practical Toolkit for Making Portable Compilers," *Communications of the ACM*, vol. 26, pp. 654-660, Sept. 1983.

Tanenbaum, A.S.: *Operating Systems: Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1987.