CHAOS

An Interactive Timeshared Operating System for the 8080

BY JEFF LEVINSKY
3632 Governor Drive
San Diego, CA 92122

CHAOS, an acronym for the Clairemont High Advanced Operating System, and its successor, CHAOS II*, are timeshared interactive systems which operate on the 8080 microprocessor. The following provides an overview of the goals of the system, examples of how CHAOS is used, and a description of the internal structure of the system, with a brief discussion of the implementation of directories and command processing. For further information, references to other articles and texts are included.

Origins of CHAOS

The initial design for CHAOS was based upon the needs of a secondary school computer course. Typical computer resources of many high schools consist of a single *port* connected to a district-owned BASIC timesharing computer or to a state or university consortium. Since only one student at a time may use the computer, each student in a class of 30 receives under nine minutes of time per week. Expansion by purchasing additional ports is limited by the high cost ($7500 is a usual price for a single port). Alternatively, microprocessor systems with multiple floppy disk drives can be obtained for under $3000, and with CHAOS, additional users can be added for about $1000 each. Moreover, an onsite computer system can be tailored to better fit the educational needs of the users. For example, the primary concern in the design for CHAOS was the availability of several languages (BASIC, 8080 assembly/machine, a system command (job control) language, PASCAL, FORTRAN, and COBOL for business-oriented students) and individual accounts for students as well as mechanisms for shared files and an 'explorable' structure. This last goal was especially chosen for the highly motivated and gifted students who frequently wish to delve into the workings of the commands and internals of the system—such activity is, in terms of CHAOS, to be encouraged, as the result is a source of knowledgeable system operators and valuable insight into system deficiencies. Of course, CHAOS is protected from non-privileged users. Security is partly maintained by the directory system, and teachers using the system to handle grading or other confidential files can elect to use a separate floppy disk for total protection. In practice, each class also has a separate floppy disk, in order to make better use of disk drives, which appear to be the most costly unit. Finally, the presence of a complete computer at the school proves far more stimulating to students, especially those interested in hardware, than does a simple remote terminal.

*Since the layout of CHAOS and CHAOS II, as opposed to the actual code, is essentially the same, 'CHAOS' will be used when no distinction between the two systems is necessary.

The UNIX^{TM} system, developed by Bell Labs for the PDP-11 computer series, served as the external model for CHAOS. Internally, UNIX contains some well-conceived and relatively simple features that CHAOS was unable to emulate, due to the limitations of the 8080 processor and the inefficiency of floppy disk storage. To produce a smaller and faster system, the majority of CHAOS is written in assembly language, as opposed to a high level language such as C. CHAOS II was developed with both hard disks and 16-bit processors such as the Z8000 in mind, so that performance can be upgraded without extensive software revision. CHAOS, like UNIX, is designed to be simple for the novice to use as well as a powerful and flexible tool in the hands of the experienced. Unlike bare-bones operating systems and exclusively high level systems, such as CP/M and UCSD PASCAL, respectively, CHAOS attempts to provide a full spectrum of computer capabilities.

A Sample Lesson

Let us imagine a student, named Fred, approaching CHAOS to write a program that will average together numbers stored in a file for one or more files. First, Fred must log onto the system (system responses are in boldface):

```
:login: fred
password:
CHAOS II
System backup at 3 pm.
You have mail.
%
```

After the password has been correctly entered (it does not echo), some system messages are printed and then the percentage sign prompt is given. Mail may be obtained by:

```
% mail
From ethel     9:30 AM  Tue Sept 5 1978
Club meeting after school.
%
```

'Mail' is an example of a *command* and is actually a non-resident program brought temporarily into Fred's work area.

Since the averaging program will be written in BASIC, that language is entered:

```
% basic
OK
```

CHAOS uses MITS BASIC Version 4.1 which provides a full set of string and file capabilities. The BASIC is actually always resident—the command 'basic' merely routes the user into it. The substance of the program is now entered:

```
10      REM     PROGRAM TO PRODUCE FILE AVERAGES
20      SUM=0          'SUM OF ENTRIES
30      COUNT=0        'COUNT OF ENTRIES
100     REM     ACCEPT FILE NAME
110     LINEINPUT "FILE?"; FILE$
120     IF FILE$="" THEN END
130     OPEN "I",1,FILE$
200     REM     SUM AND COUNT ENTRIES
210     IF EOF(1)  THEN CLOSE : GOTO 300
220     INPUT #1, NUM
230     SUM=SUM+NUM : COUNT=COUNT+1
240     GOTO 200
300     REM   OUTPUT
310     IF COUNT=0 THEN PRINT "NO ENTRIES" ELSE PRINT SUM/COUNT
320     GOTO 100
```

and tested:

```
RUN
FILE? DATA1
32.778
FILE? DATA2
29.14
FILE?
OK
```

Now, the program is stored (in Fred's directory by default) and BASIC is exited:

```
SAVE "AVERAGE"
OK
CHAOS
%
```

Fred can now check that the file indeed is in his directory via the command 'dir':

```
% dir
average
data1
data2
master
stuff
%
```

More information about the contents of the directory could be obtained by

```
% dire  -al
```

where the 'a' and 'l' are flags interpreted by 'dir' as a signal to provide special options. Commands may inspect flags and other arguments typed upon the command line at will. In this case, 'a' causes all filed in the directory to be listed normally files with names beginning with a period are not printed out) and 'l' causes length, access, and disk information to be given for each file. At any rate, Fred can run his new program without re-entering BASIC:

```
% AVERAGE
FILE? DATA2
29.14
%
```

Here, AVERAGE was run just as a command—which is not surprising given that the commands 'mail,' 'basic,' and 'dir' are all actually written in BASIC.

The argument facilities exist as *system calls* and are available in BASIC to privileged users. CHAOS offers an all-or-none policy in this area: either one has or does not have access to BASIC commands such as POKE and OUT and the ability to make system calls. The right to use these is known on CHAOS as 'FRIBL' (pronounced 'fribble'). A CHAOS command such as 'dir' may have 'FRIBL' although the user who executes the command does not. One way that Fred may obtain 'FRIBL' is to move from his directory into the directory 'master,' in which a special version of the command 'basic' exists. Since Fred has a pointer to 'master' (see the results of 'dir' above), he transfers directories by:

```
% chdir master
```

and then enters the version of BASIC which gives him 'FRIBL.' Although in BASIC, Fred can still transfer back to his own directory, which is what '. . . ' refers to here*:

```
% frbasic
OK
chdir ". . . "
OK
```

The modified version of AVERAGE that uses the argument facility is:

```
20      SUM=0          'SUM OF ENTRIES
30      COUNT=0        'COUNT OF ENTRIES
40      ARG=2          'ARGUMENT NUMBER
50      DIM M(5)       'ARRAY FOR SYSTEM CALLS
60      FRIBL          'THIS COMMAND NEEDS FRIBL
100     REM    ACCEPT FILE NAME
120     GOSUB  9000

320     END

9000    REM    READS ARGUMENT #ARG INTO FILE$
9010    M(0)=36        'SYSTEM CALL NUMBER
9020    M(1)=ARG
9030 X=USRO(VARPTR(M(0) ) )
9040 X-1  :  FILE$= " "
9050 FILE$=FILE$ + CHR$(PEEK(M(4) ) AND 127)
9060 IF(PEEK(M(4) ) AND 128) = 0 THEN M(4)=M(4)+1 : GOTO 9050
9070 RETURN
```

Lines 130 through 310 are the same as before. The subroutine at 9000 reads in the argument ARG from the command line into the string FILE$. There are several ways to accomplish this. Here the pointer to the start of the argument returned by the system call in M(4) is used to PEEK out the characters (the most significant bit being on only in the last character of the argument). FRIBL at line 60 can only be entered here because Fred already has FRIBL.

*Paths are dynamic in CHAOS, as opposed to static on UNIX.

Again, the program is stored and run back at the system level:

```
SAVE "AVERAGE"
OK
CHAOS
% AVERAGE DATA2
29.14
%
```

Now, however, averaging is only accomplished for a single file. To permit several files to be averaged, one could write a loop in BASIC to increment ARG (null arguments can be tested for), but CHAOS itself provides an alternative method. This consists of writing a file of commands known as a *"shell"* file (the command line processor is known as the *"shell"*) to make use of some argument processing commands. In this case, Fred produces the shell file 'AVERAGER' which contains the following text lines:

```
:loop
if $2a a  ; exit
echo $2
average $2
shift ; goto loop
```

When the shell file is executed, each command inside is executed in turn. The first command here begins with a colon, and serves only as a place holder for the label 'loop.' The 'if' command in the next line compares two strings/arguments for equality, in this case the '$2a' against the 'a.' A dollar sign followed by a number is a macro, and so '$2a' will be replaced by the second argument on the command line with AVERAGER with an 'a' appended at the end. If this is a null argument, then '$2a' will match 'a' and thus the exit command on the same line will be executed causing the whole shell file to terminate. If some argument '$2a' does exist, then 'exit' will not be executed. On the next line, 'echo' is used simply to print the second argument out, and after that, the average of the file is printed. 'Shift' causes arguments to be shifted to the left, so that what was the third argument to AVERAGER will become the second. Finally, the 'goto' will cause a jump back to the first line for another averaging.

To use AVERAGER, Fred merely says:

```
% AVERAGER DATA2 DATA1
DATA2
29.14
DATA1
32.778
%
```

At this point, Fred may wish to have AVERAGE and AVERAGER moved into the root directory so that all users can access them. Then, by typing a control -d, Fred exits from the system:

```
% d
:login:
```

A few observations can be made here. From the user's viewpoint, the command developed above is convenient as the standard system-wide format for the arguments is easy to remember. Having developed a command that becomes avail- to all users, Fred is motivated to continue to produce and to upgrade. Meanwhile, other programmers can readily incorporate AVERAGE or AVERAGER into their commands using the capabilities of either BASIC or the SHELL. CHAOS also permits commands to be written in native code, and these may also invoke other commands and make system calls. UnFRIBLed users may run and debug 8080 machine code, but only under the watchful eye of a simulator which prevents system crashes. Finally, if Fred is on a multi-user version of CHAOS, other users may at the same time be running and debugging commands/programs independently. Of course, the system has safeguards to prevent file modification from occurring while others are using the same file.

The Structure of CHAOS II

CHAOS II consists of about 10K of code which resides in the lower 32K of memory alongside the 20K MITS BASIC. The system is both conceptually and physically divided into a series of nested levels, which are diagrammed in Figure 1. The contents of a given level may call upon an inner level for assistance, but the reverse is not permitted. For example, the lowest level is the 8080 hardware: this is used by all other levels but cannot itself use any of the software in those levels. One may be reminded of a black hole, into which any object may be sent, but from which no object will emerge. Each level other than level 0 consists of a number of carefully defined system calls, such as the one used above to obtain a pointer to a shell argument. Typically, each level is written, tested, and documented by one person. Briefly, these levels are:

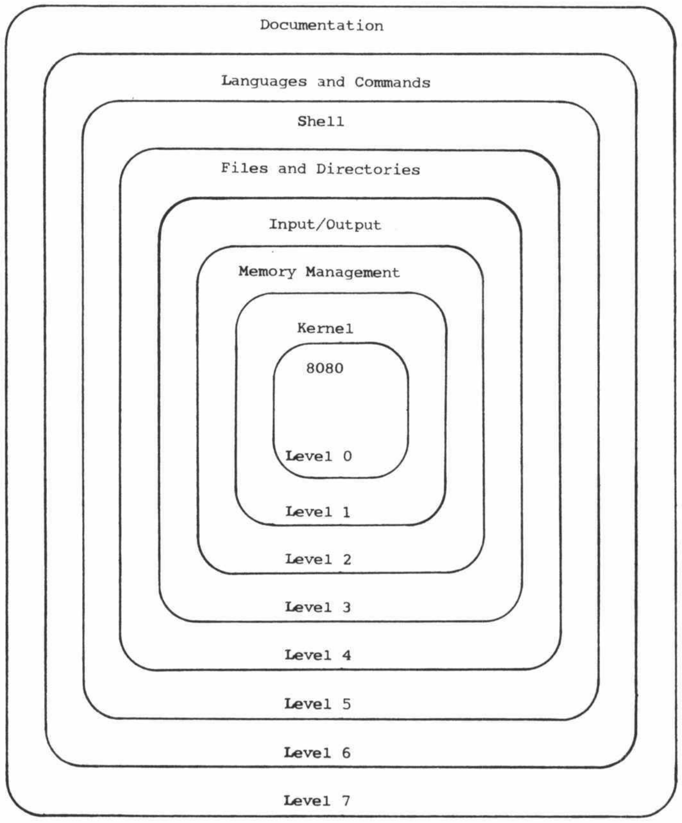

Figure 1. CHAOS System Structure

Level 0: The 8080 hardware required to run CHAOS can consist of no more than a standard MITS 8080 floppy disk system with about 40K of memory, if only a single-user version is required. The system is configured as shown in Figure 2 with BASIC and CHAOS in the low 32K, the user's system variables in the 33rd K of memory, and with the remainder serving as the user's workspace.

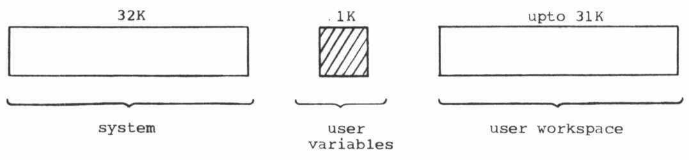

Figure 2. Single User Configuration

If multiple users are to be supported, then one of two configurations is possible. In the first, shown in Figure 3, a separate 1K variable area is provided for each user and all workspaces are located in the single upper block of memory. All of the variable areas are addressed the same but only one may be active at a given time.

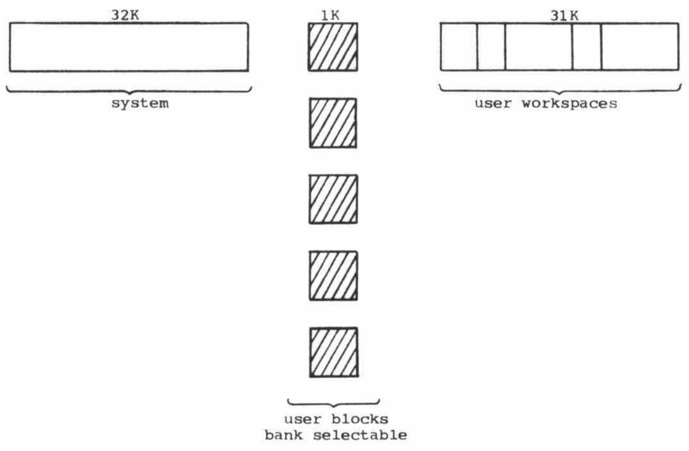

Figure 3. Multi-user configuration

This is controlled by a signal output to a parallel port which in turn selects one of eight 'banks' from a memory board, as is shown in Figure 4. The front board in that figure is active only when the low 1K of the boards is being addressed – otherwise the rear board is enabled.

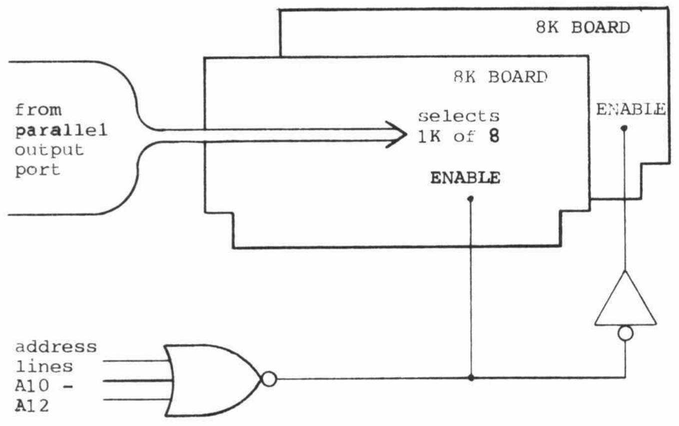

Figure 4. Multi-user control hardware

When large workspaces are required, the configuration given in Figure 5 is used. Under this method, both the 1K variable spaces and the workspaces are selected by the signal sent to the port. The hardware to implement this is shown in Figure 6, and consists only of a decoder connected so as to disable all but one set of 16K boards, i.e., all but those of the active user.

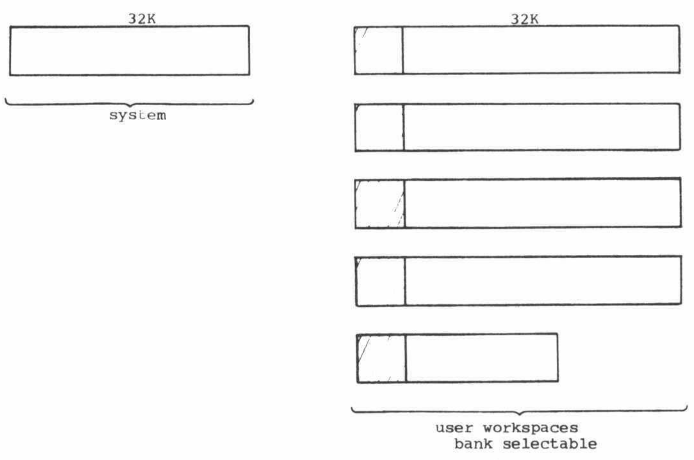

Figure 5. Multi-user configuration

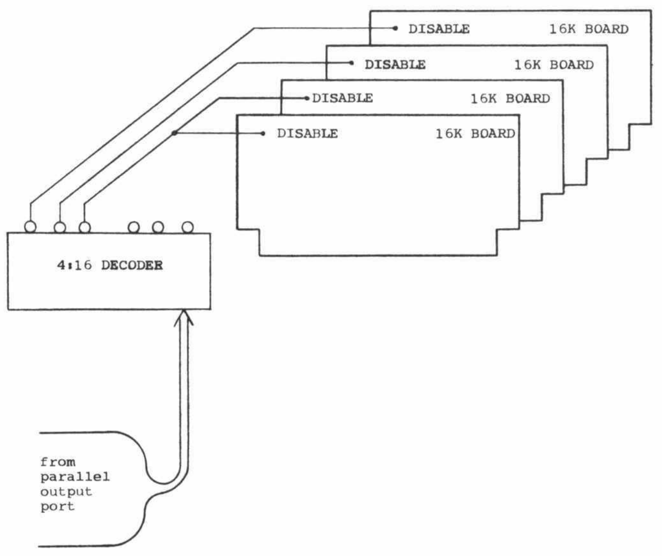

Figure 6. Multi-user control hardware

Level 1: The *kernel* provides each user/process with a slice of computer time, maintains the system clock, and provides some interprocess synchronization. The kernel handles interrupts and controls the memory switching described above for multi-user systems. Within the kernel, a circular queue of active processes is kept for use in scheduling. If a process requests a semaphore (a P operation) when the semaphore is currently unavailable, the process is suspended from the circular queue and placed upon a FIFO queue associated with the semaphore. When another process then releases the semaphore (a V operation), a process from the corresponding semaphore queue is then restored to the active queue.

Level 2: Memory management consists of the allocation of *segments* within each user's workspace. The layout of the typical workspace is shown in Figure 7: a stack of segments grows upwards from the bottom of the workspace while the 8080 stack grows downwards from the top. Each segment is either a program/command or a special data area required by a program. The segments are handled similarly to 'activation records' in programming languages and system calls exist to create, modify, and remove these.

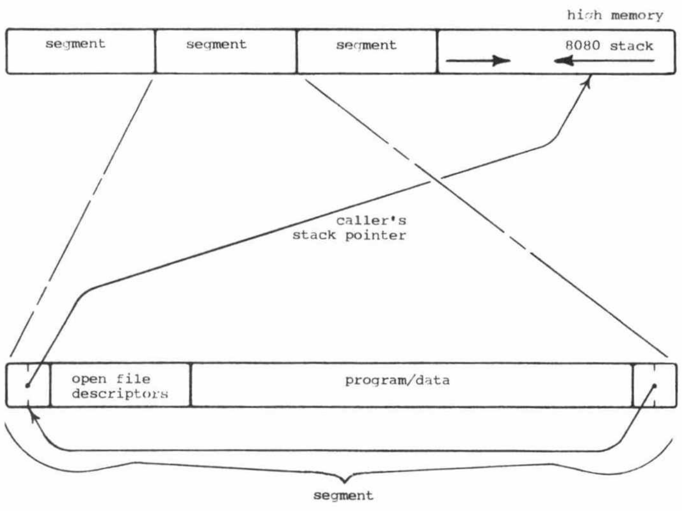

Figure 7. User's workspace and segments

Level 3: Input/output constitutes a minor part of CHAOS as complex buffering and device drivers are avoided. At least in secondary schools, students enter data relatively slowly and do not want blindingly fast screen listings. Therefore, no tremendous efficiency is required. In the multi-user version of CHAOS II, vectored interrupts can be added to achieve somewhat smoother input/output, especially as disk head motion need not lock onto the processor.

Level 4: Files and directories are quite elaborate in CHAOS as evidenced by the example given above. Essentially, a directory is simply a file of pointers to other files, and therefore directories may point to other directories (as 'fred' pointed to 'master'). In CHAOS II, file and directory functions are primarily handled by modified code from the BASIC.

Level 5: The Shell was already introduced as the program responsible for processing command lines. How it operates is explained more thoroughly below, but its position in the hierarchy is clear here, for it must rely upon input/output to receive the commands from the user, upon memory management to create and remove the segments to hold the commands, and upon the file and directory system to locate and to read in the commands. The Shell is probably the most independent of all the levels described so far, and certainly other shells which act in quite different ways can be used. In fact, halfway through the debugging of the original CHAOS, one shell was pulled out and another substituted in for it without more than a handful of errors.

Level 6: Languages and commands are properly one level outside of the Shell, although parts of BASIC occur at inner levels as much of the BASIC code is used for input/output and for files. Several options exist for other languages: a Tiny-LISP and the 8080 simulator fit into a user's workspace under all configurations, while large languages require a full 31K workspace as do most business application packages.

Level 7: Documentation is of crucial importance on CHAOS since users have a variety of backgrounds and interest. After the UNIX fashion, each command and system call is given a one or two page summary which provides adequate information in most cases. Commonly used programs, such as the text formatter (word processor) and the BASIC are also described by tutorials which emphasize examples that users may actually type in and try. On a larger scale, the system is described by a set of charts which show the flow of control in operations such as argument parsing and directory searching. Finally, a current project is the completion of a manual detailing the overall operation of the system through diagrams and text, of which a condensed and simplified version will be provided to users seeking only general knowledge. CHAOS itself is expected to assist in documentation both by facilitating the production of manuals and updates and by providing a complaint facility through which frustrated command users may communicate with those responsible for maintaining the system.

The CHAOS Directory System

Each file on CHAOS is described by a *file pointer* which can be defined in PASCAL as:

```
type
    ftype = (basic, text, assembly, random, directory);
    rights = (w, o, r, e);
    filepointer = record
        name:           array[1..8] of char;
        origin:         integer;     [sector and track #]
        access:         set of rights;
        owner:           filepointer:
        lastwritten:    date;
        case oftype:    ftype of
          basic, text, assembly, random: (length: integer)
          directory:                      (quota: Integer)
        end;
```

With this, we might define a directory as:

```
    directory = set of filepointer;
```

(although this isn't legal PASCAL). How directories are actually implemented is not of importance: in CHAOS both files of descriptors and a common single disk directory (volume table of contents) have been used.

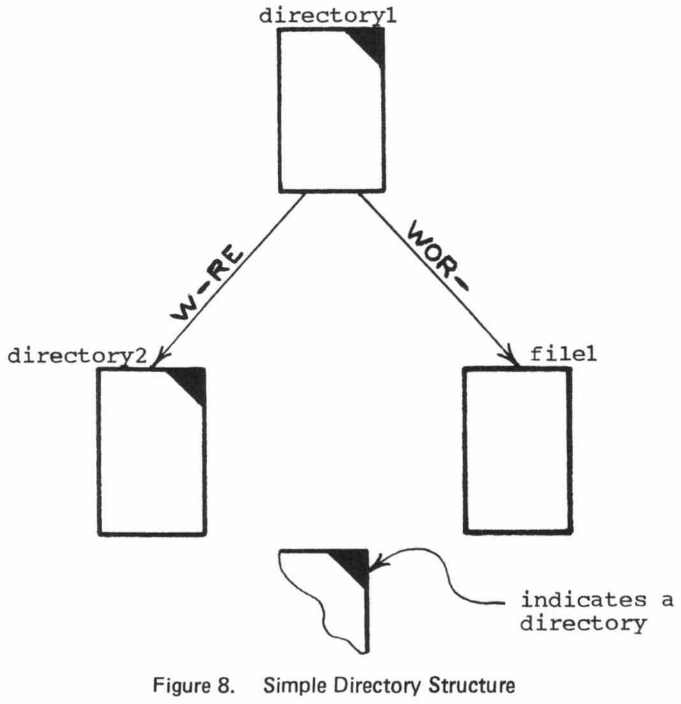

Figure 8. Simple Directory Structure

Mainly to introduce some notation, Figure 8 shows 'directory1' which owns two entries: 'directory2' and 'file1'. On each arrow are the access rights for the entry. These govern what operations a user in 'directory1' can perform upon the entries. The four possible access rights are:

w = write access
r = read access
e = execute access
o = override access

Having override access permits any operation to be performed. Access rights are also associated with a user in a given directory. To actually compute what access is permitted for an entry, the access rights associated with the owning directory are *anded* with those associated with the entry itself. This is best illustrated by example. Suppose that a user has r and e access when in 'directory1'. If that user accesses 'file1', then the rights are computed as:

```
--re
wor-
--r-
```

The file can only be read. On the other hand, if the user has just o access for 'directory1', then the rights computed for 'file1' are:

```
-o--
wor-
-o--
```

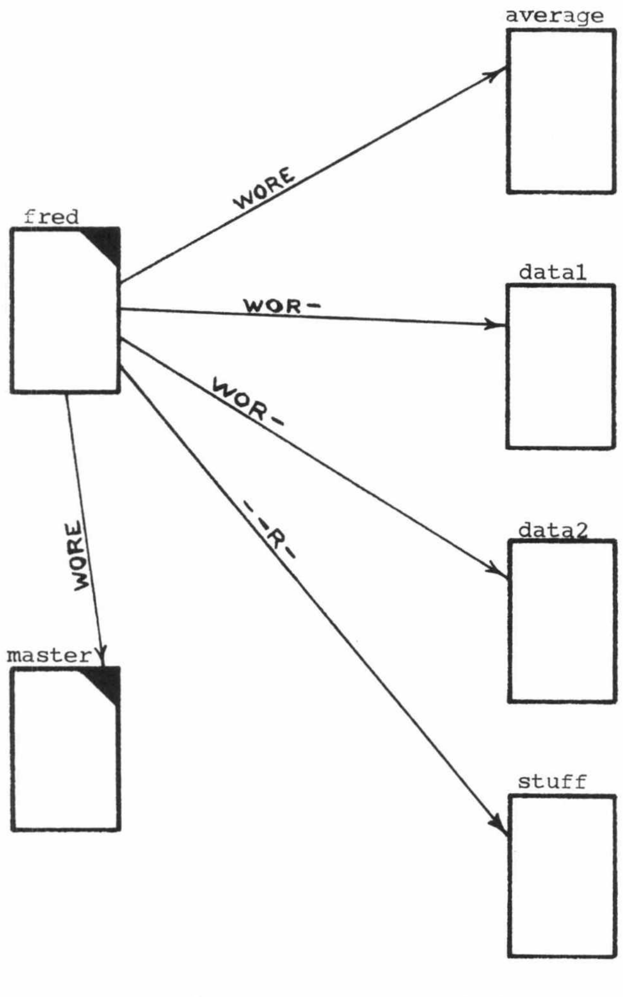

Figure 9. Directory structure for 'fred'

Here, since override access exists, all operations can be performed upon the file. When a user elects to change from one directory to another, the user's rights for the new directory are computed in the same manner. Supposing that a user has o and r access in 'directory1', the access rights for the user upon transfering into 'directory2' will be computed as:

```
-or-
w-re
--r-
```

When the user returns to 'directory1', the o and r status will be restored.

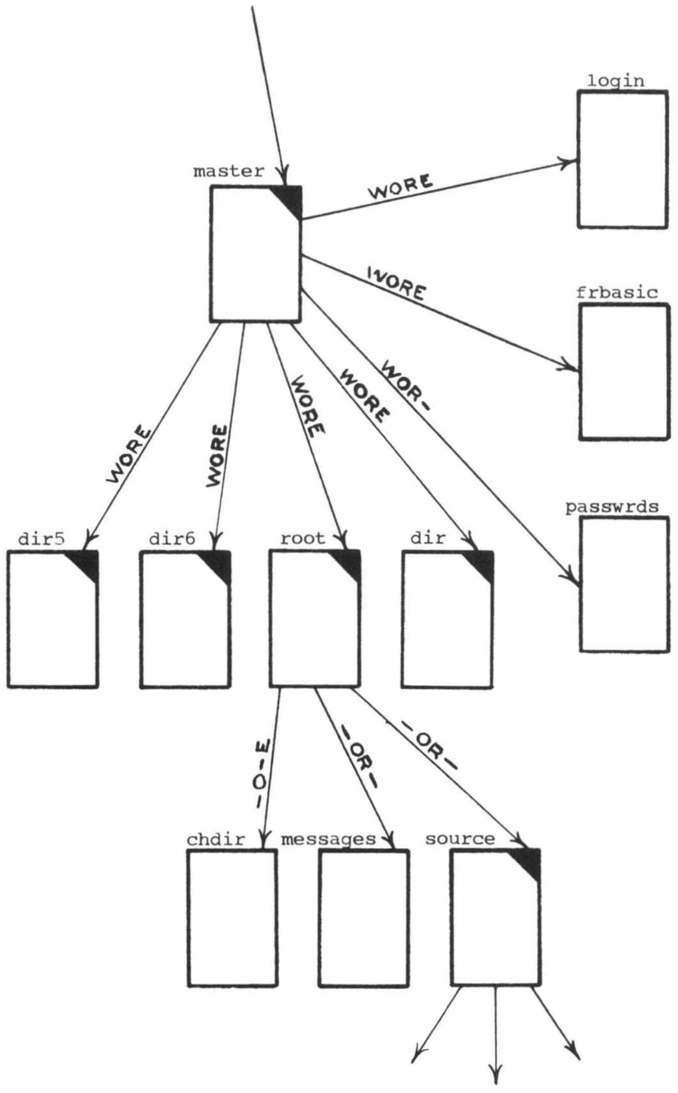

Figure 10. Directory structure for 'master' and 'root'

Figure 9 illustrates the directory 'fred' used in the sample session above. Since the data files are not generally executable, they are by default not given e access. Assuming that Fred had o access when in his directory, those data files could still have been executed by him—in which case they are treated as shell files. When Fred moves into 'master', he retains all possible rights. Figure 10 shows the typical structure of 'master'. The program 'login' controls the actual logging-on of users into CHAOS and accesses the passwords file. The directory 'master' then also contains pointers to all directories into which users may intially enter.

Since file pointers are constrained to point only to files on their disk (to allow single disks to be unmounted). 'master' is actually spread across all disks. One other directory of importance is 'root' as it contains all of the typically used system commands. The Shell first searches a user's directory for a file to execute and then, if the file is not found, the 'root' directory. Each user has an implicit pointer to 'root' with w, r, and e access. Hence, a user referencing the command 'chdir' when not in 'root' will have access rights computed as:

```
w-re
-o-e
---e
```

and will only be able to execute. In this example, the user will have only r access to the directory of source code and to the messages file. On the other hand, a user transferring into or logging into 'root' via 'master' will have full access rights due to the presence of the override access in all of the pointers within 'root.' Presumably, if a user turns off all rights on a pointer to a directory and then transfers into that directory, no operations will be possible. For this reason, the implicit access to 'root' is fixed rather than computed: the user will always be able to call 'chdir' in order to return to the original directory.

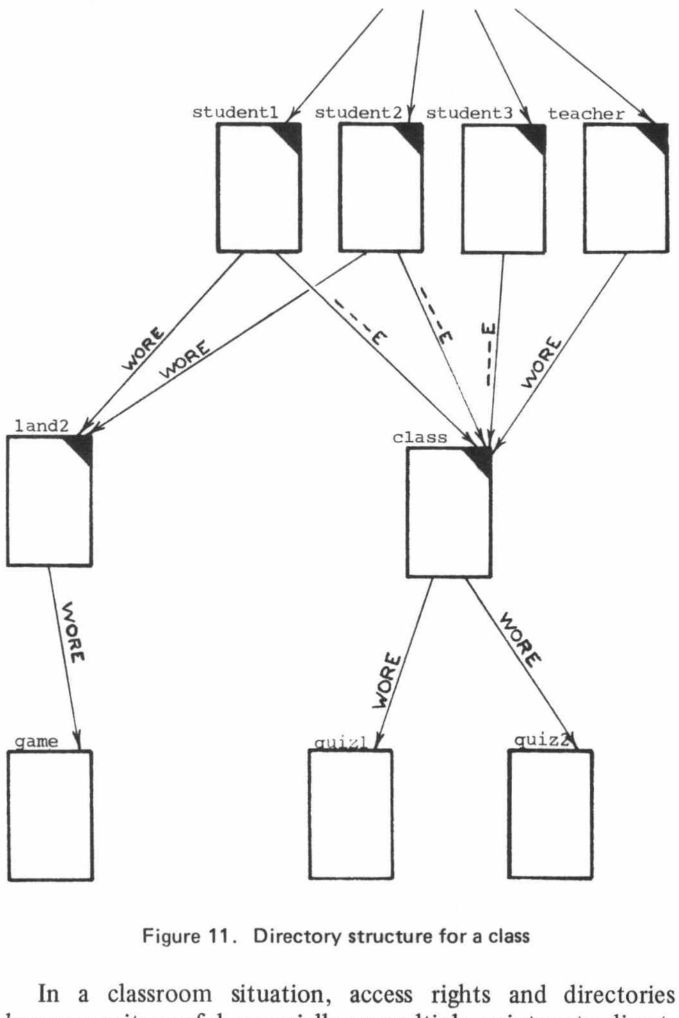

Figure 11. Directory structure for a class

In a classroom situation, access rights and directories become quite useful, especially as multiple pointers to directories may exist (multiple pointers to regular files are not typically permitted, to avoid the issue of choosing which owner is to be charged for the file's length). A common directory for a class is shown in Figure 11. The directory 'class' is akin to 'root' in that the students may only execute while the teacher has full access to the quiz files. The directory 'land2' exists so that two students may both work on a game program that is protected from other users. Of course, a user with access to 'master' has access to all directories under normal circumstances.

Implementation of the Shell

No doubt the most interesting feature of CHAOS, due to its command and argument processing abilities, is the Shell. Strictly speaking, the Shell is a function which processes one or more command lines to obtain a boolean result indicating the success or failure of the commands. The 'if' command presented earlier is capable of using the boolean value returned as a conditional, as are assembly language programs. A few variants of the function shell exist: shellf executes a shell file, and shelli produces an interactive shell (one that prints prompts and accepts command lines from the terminal—exiting only after a control-d is typed).

Since a shell file may, for example, invoke another shell file or an interactive shell, the shell function is recursive. The command lines and/or pointers into them are kept within a segment created for a specific invocation of the shell function. A segment associated with an interactive shell is dynamic in size as command lines inputted vary in length. A typical interactive shell segment is shown in Figure 12. Shell segments also contain a list of processed arguments from the command line currently being executed. These arguments can be accessed in various ways by the system calls contained in the Shell level of the system.

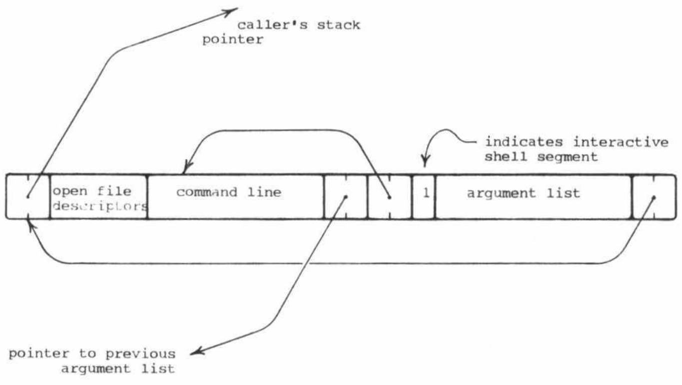

Figure 12. Interactive shell segment

The general algorithm for the shell function is very straightforward:

```
function shell (commandline: array [1 . .n]  of char):
 boolean;
   status: boolean;
begin
   status:=true;
   repeat
      process command line;
      if legal then status:= shell(command)
          else status:=false
   until endoffile or (status = false);
   shell:=status
end;
```

The function shelli is slightly different in that it prints a prompt and requests a new line whenever an end-of-line is reached or an error in a command line is found. Non-interactive shells return upon an error: therefore when a shell file is found to be incorrect, execution of it will terminate and control will be returned to an interactive shell or machine language program. This may cause intermediate shell files to also terminate—the flow of control is handled automatically by the shells.

About half of the Shell is devoted to command line argument processing and especially to the generation of argument lists. In general, an argument is a string of characters delimited by blanks or other special characters. The command 'chdir master' consists of two arguments: the first argument is taken of course to be the name of the file to be executed. Arguments may contain special characters through the use of quoting, which entails either surrounding the argument with single or double quote marks or by turning off the special meaning of a character by preceding it by a backslash. For example, to run the command whose name is ';', one might say:

```
% ';'     or     % " ; "     or     % \;
```

however '% ; ' would not work as the semicolon would be taken as a command separator (in this case separating two null commands).

In addition to quoting and macros, CHAOS' Shell permits what is often called argument list generation or wild-character file names. Whenver certain characters are found (unquoted) in an argument, the Shell no longer treats the argument literally but instead takes it to be a pattern to be matched against the names of all files in the current directory. Those names which match are then included into the argument list (in alphabetic order). The special characters and their meanings are:

?	matches any single character.
[	defines the beginning of a character class.] defines the end. Characters appearing between the square brackets are all potential matches for a single character in a file name. Hence ' [abcd] ' would match a single 'a', 'b', 'c', or 'd'. A shorthand for this is '[a-d] '. If the character directly following the '[' is a dash, then the character class matches any character *but* those appearing inside (i.e., the intial dash negates the meaning). Since ' [ ] ' matches no character, ' [-] ' matches all and is equivalent to '?'.
*	signifies that zero or more of the previous character or character class may be matched.

Some examples may clarify how file names can be selected using these patterns. Say that the current directory contains the files 'data1','data2', 'data33', and 'datum'. Then typing:

```
% print data?
```

is equivalent to typing:

```
% print data1 data2
```

as the question mark matches the '1' and then '2'. Neither 'datum' nor 'data33' was included in the list because the 'u' or the final '3' did not match. On the other hand,

```
% print dat??
```

is equivalent to:

```
% print data1  data2  datum
```

To get all file names of the form data⟨digit⟩, one might use 'data[0-9] '. This would not match a file named 'datam' for example. To also match 'data33', one might use 'data [0-9] *'. The asterisk applies in this case to the character class and causes zero or more digits to be matched, that is, both 'data' and 'data1234567' will also match this pattern. More generally, '?*' will match any file name (the question mark is optional in this case).

To process a command line, Shell repeatedly calls a scan function which in turn examines the command line for a single argument, appends one if found to the argument list at the end of the shell segment, and then reports back via a flag whether or not the end-of-line has been located. The scan function identifies quoted arguments, tracks down macros, and supervises argument expansion. The actual procedure to compare a pattern to a file name is somewhat complex, as recursion is required to handle patterns such as 'a*b*', and is based upon programs published in *Software Tools* (see References).

Finally, given the directory structure and shell calls outlined above, the actual external appearance of the system can be very easily described. The 'login' command in the master directory prints the request for a directory name and a password, and then determines whether such a directory exists and if the password is correct—using the pointers from 'master' to establish the user's original directory and access rights. Thus, each user is initialized to run the following:

```
program chaos;
  const       crash=false;
begin
  repeat
    shell("login");
    shelli
  until crash
end
```

References

For information on CHAOS and CHAOS II, write to Mary Elizabeth Kroening, 3632 Governor Drive, San Diego, California, 92122. Persons interested in specific applications of CHAOS II, such as in education, medical research, or business should ask for details on the high-level system packages being developed for these areas.

Brooks, Frederick P., Jr. *The Mythical Man-Month.* Menlo Park: Addison-Wesley Publishing Company, 1975.

Gries, David. *Compiler Construction for Digital Computers.* New York: John Wiley & Sons, 1971.

Kernighan, Brian W., and Plauger, P.J. *Software Tools.* Menlo Park: Addison-Wesley Publishing Company, 1976.

Ritchie, Dennis M., and Thompson, Ken. "The UNIX Timesharing System." *Communications of the ACM,* XVII 7 (July, 1974), 365-375.

Shaw, Alan C. *The Logical Design of Operating Systems.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974.

Wirth, Niklaus, and Jensen, Kathleen. *PASCAL User Manual and Report.* New York: Springer-Verlag, 1975.