A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.
- No operating system changes are required to install or use the system.
- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.
- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

†UNIX is a Trademark of Bell Laboratories.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Purpose

The widespread use of the UNIX† system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[2,3] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster

†UNIX is a Trademark of Bell Laboratories.

than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[4,5] Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp* or *uux* programs,

c) by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number,

[6] login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S send a file,
R receive a file,
C copy complete,
H hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in

and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.[6] Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[7] There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.
2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.
2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.
3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.
4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

Acknowledgements

We thank G. L. Chesson for his design and implementation of the packet driver and protocol, and A. S. Cohen, J. Lions, and P. F. Long for their suggestions and assistance.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).
2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* 57(6) pp. 2177-2200 (1978).
3. G. L. Chesson, "The Network UNIX System," *Operating Systems Review* 9(5) pp. 60-66 (1975). Also in *Proc. 5th Symp. on Operating Systems Principles.*
4. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications*, p. 21F (June 1974). IEEE Cat. No. 74CH0859-9-CSCB.
5. A. G. Fraser, "A Virtual Channel Network," *Datamation*, pp. 51-56 (February 1975).
6. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey (June 1976).
7. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, Mass. (1975).

Uucp Implementation Description

D. A. Nowitz

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current implementation of uucp. Is is designed for use by an administrator/installer of the system. It is not meant as a user's guide.

Introduction

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. It can be used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execute files* are scripts for UNIX commands that involve the resources of one or more systems.

There are four primary programs:

uucp	builds *work files* and gathers *data files* in the spool directory for data transmission.
uux	creates *work files*, *execute files*, and gathers *data files* for the remote execution of UNIX commands.
uucico	executes the work files for data transmission.
uuxqt	executes the scripts for UNIX command execution.

There are a couple of administrative programs:

uulog	gathers temporary log files that may occur due to lockout of the uucp log file and reports some information such as copy requests and completion status.
uuclean	removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

---

† UNIX is a Trademark of Bell Laboratories.

1. M. E. Lesk and A. S. Cohen. UNIX *Software Distribution by Communication Link*, private communication.

1. Uucp — UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The command is designed to look like *cp* to the user. The syntax is

uucp [option] ... source ... destination

where the source and destination may contain the prefix *system-name!* , which indicates the system where the file or files reside or where they will be copied.

Uucp has several options:

- −d Make directories when necessary for copying the file.
- −c Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.
- −e*sys* Send this job to system *sys* to execute. (Note that this will only work when the system *sys* allows *uuxqt* to execute a *uucp* command. See the "Uuxqt" and "Security" sections.)
- −g*letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)
- −m Send mail to the requester on completion of the work.
- −n*user* Notify *user* on the remote machine that a file has been sent.

There are several options available for debugging:

- −r Queue the job but do not start *uucico* program.
- −x*num* *Num* is a level number between 1 and 9; higher numbers give more debugging output.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. If the directory exists, it must be writable by everybody. (Note that if the destination is a directory name and the "−d" option is specified to create the directory, the directory name must be followed by "/".) The source name may contain special shell characters such as "?*[]". These will be expanded on the appropriate system.

The command

```
uucp  *.c  usg!/usr/dan
```

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the "usg" machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory of *user* on the specified system. File names beginning with "~/" translate into the public directory (usually /usr/spool/uucppublic) on the remote system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted for security reasons.

The command

```
uucp  usg!~dan/*.h  ~dan
```

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names, the system-part of each argument, and the options to classify the work into several types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote system.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

[6] Request that the *uucp* command be executed by a remote system.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine and execute the work (unless the – r option was specified).

Type 1 - local copy

The copy is done locally. The *–m* and *–n* options are not honored in this case.

Type 2 - receive files

A *work file* is created or appended with a one line entry for each request. The upper limit to the number of files per *work file* is set in *uucp.h*. (The default setting is 20.) After the limit has been reached, a new *work file* is created. (All *work files* and *execute files* use a blank as the field separator.) The fields for these entries are given below.

[1] R

[2] The full path-name of the source or a ~something/path-name. The ~*something* part will be expanded on the remote system. .

[3] The full path-name of the destination file. If the ~*something* notation is used, it will be immediately expanded.

[4] The user's login name.

[5] A "– " followed by an option list. The options – m and – d may appear.

Type 3 - send files

Each source file is copied into a *data file* in the spool directory. (A "– c" option on the *uucp* command will prevent the *data file* from being made. In this case, the file will be transmitted from the indicated source.) The fields for these entries are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ~something/file-name.

[4] The user's login name.

[5] A "– " followed by an option list. The options – d, – m, and – n may appear.

[6] The name of the *data file* in the spool directory. A dummy name, "D.0" is used when the – c option is specified.

[7] The file mode bits of the source file in octal print format (e.g., 0666).

[8] The user on the remote system to be notified upon completion of the file copy when the "– n" option is specified.

Type 4 and Type 5 - remote uucp required

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

Type 6 - remote execution

This occurs when the "– e" option is used. In this case, the *uux* facility is used to create and send the request. This requires that the remote *uuxqt* program allows the *uucp* command.

2. Uux–UNIX To UNIX Execution

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

uux [–] [option] ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as "< >|^" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments that do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) An argument that contains a "!" but is not to be treated as a file at the present time, can be escaped by using "()" around the argument. (Note that the "()" symbols must usually be escaped with a "\" symbol.) The "–" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The following options are available for debugging:

– r Don't start *uucico* or *uuxqt* after queuing the job.

– x*num* *Num* is a level number between 1 and 9; higher numbers give more debugging output.

The command

pr abc | uux – usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

Uux generates an *execute file* that contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a send command (type 3 above).

For required files that are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine for execution by the *uucico* program.

The *execute file* contains a script that will be processed by the *uuxqt* program. It is made up of several lines, each of which contains an identification character and one or more arguments. The lines are described below.

User Line

U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

F file-name real-name

where the *file-name* is a unique name used for file transmission and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present. The *uuxqt* program will check for the existence of all these files before the command is executed.

Standard Input Line

I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "–" option is used. If a standard input is not specified, "/dev/null" is used. (Note that if there is a standard input specified, it will also appear in an "F" line.)

Standard Output Line

O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note that the use of ">>" is not implemented.)

Command Line

C command [arguments] ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (usually /usr/lib/uucp/.XQTDIR) and the UNIX command is executed using the shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program. (Note that a check is made to see that the command is allowed as specified in the *uuxqt* program.) After execution, the standard output is copied or sent to the proper place.

3. Uucico—Copy In, Copy Out

The *uucico* program will perform several major functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways:

a) by a system demon specified in a crontab entry,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the "shell" field in the "/etc/passwd" file for the logins used by remote systems to access uucp.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (−s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

Uucico is generally started by another program. There are several options used for execution:

−r1 Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.

−s*sys* Do work only for system *sys*. If *−s* is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems that do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

−d*dir* Use directory *dir* for the spool directory.

−x*num* *Num* is a level number between 1 and 9; higher numbers give more debugging output.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

type . system-name grade number

where

type is an upper case letter (*C* – copy command file, *D* – data file, *X* – execute file),

system-name is the remote system,

grade is a character,

number is a four digit, zero padded sequence number.

The file

C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call Remote System

The call is made using information from several files that reside in the uucp program directory (usually /usr/lib/uucp). At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The *L.sys* file contains information required to make the remote connection:

[1] system name,

[2] times to call the system (days-of-week and times-of-day) and the minimum time delay before retry,

[3] device or device type to be used for call,

[4] line class (this is the line speed on almost all systems),

[5] phone number if field [3] is *ACU* or the device if not *ACU*,

[6] login information (zero or more fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g., mh, py, boston) that get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try each devices that satisfy [3] and [4] until a call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created. If the call is completed, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where *proto-list* is a string of characters, each representing a line protocol.

The calling program checks *proto-list* for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where *code* is either a one character protocol letter or "N", which means there is no common protocol.

Work Processing

The *MASTER* program does a work search similar to the one used in the "Scan For Work" section. (The *MASTER* has been specified by the "– r1" uucico option.) Each message used during the work processing is specified by the first character of the message:

- S send a file,
- R receive a file,
- C copy complete,
- X execute a *uucp* command,
- H hangup.

The *MASTER* will send *R*, *S* or *X* messages until all work for the remote system is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY*, *SN*, *RY*, *RN*, *HY*, *HN*, *XY*, or *XN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the spool directory to the destination. Otherwise, a *CN* message is sent. (In this case, the file is put in the public directory, usually /usr/spool/uucppublic, and the requester is notified by mail.) The requests and results are logged on both systems.

The hangup response is determined by a work scan of the *SLAVE*'s spool directory. If work for the remote system exists an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems unless a "– s" option was specified.

4. Unxqt—Uncp Command Execution

The *uucqt* program is used to execute scripts generated by *uux*. The *uucqt* program may be started by either the *uucico* or *uux* programs or a demon specified by a *crontab* entry. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line is verified and executed.

The *execute file* is described in the "Uux" section above.

The execution is accomplished by executing a "sh – c" of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

5. Uulog—Uucp Log Inquiry

When a *uucp* program can not make a log entry directly into the *LOGFILE* an individual log file is created: a file with prefix *LOG*. This will sometimes occur when more than one uucp process is running. Periodically, *uulog* may be executed to append these files to the *LOGFILE*.

The *uulog* program may also be used to request the output of *LOGFILE* entries. The request is specified by the use of the options:

- −s*sys* Print entries where *sys* is the remote system name,
- −u*user* Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

6. Uuclean—Uucp Spool Directory Cleanup

This program is typically started by the uucp daily demon. Its function is to remove files from the spool directory that are more than 3 days old. These are usually files for work that can not be completed. The requester of this work is notified that the files have been deleted.

There are several options:

- −d*dir* The directory to be scanned is *dir*.
- −m Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. This mail is sometimes useful for administration.)
- −n*hours* Change the aging time from 72 hours to *hours* hours.
- −p*pre* Examine files with prefix *pre* for deletion. (Up to 10 of these options may be specified.)
- −x*num* This is the level of debugging output desired.

7. Security

☞ *The uucp system, left unrestricted, will let any outside user execute any commands and copy out/in any file that is readable/writable by a uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.*

There are several security features available aside from the normal file mode protections. These must be set up by the administrator of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started so that all work is done through *uucico*.
- The owner of the uucp programs should be an administrative login. It should not be one of the logins used for remote system access to uucp.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the "Files Required For Execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident of the caller's identity.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specified in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by the uucp administrative login and have mode 0400 to protect the phone numbers and login information for remote sites.

– The programs *uucp*, *uucico*, *uux*, *uuxqt*, *uulog*, and *uuclean* should be owned by the uucp administrative login, have the setuid bit set, and have only execute permissions.

8. Uucp Installation

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user or the uucp administrative login. However, several remote computers may use the same login name.

Each computer should be given a unique *system name* that is transmitted at the start of each call. This name identifies the calling machine to the called machine. The *login/system* names are used for security as described later in the *USERFILE* section.

There are several source modifications that may be required before the system programs are compiled. These relate to the directories, local system name, and attributes of the local environment.

There are several directories used by the uucp system:

lib	(/usr/src/cmd/uucp) – This directory contains the uucp system source files.
program	(/usr/lib/uucp) – This is the directory used for some of the executable system programs and the system files. Some of the programs reside in "/usr/bin".
spool	(/usr/spool/uucp) – This is the uucp system spool directory.
xqtdir	(/usr/lib/uucp/.XQTDIR) – This directory is used during execution of the *uux* scripts.

The names in parentheses above are the default values for the directories. The italicized names *lib*, *program*, *xqtdir*, and *spool* will be used in the following text to represent the appropriate directory names.

There are two files that may require modification, the *makefile* file and the *uucp.h* file. (On some systems, the makefile is named *uucp.mk*.) In addition, the "uuxqt.c" program may be modified as indicated in the "Security" section above. The following paragraphs describe the modifications.

uucp.h modification

Several manifests in "uucp.h" may need modification for the local system environment:

UNAME	should be defined if the "uname" function is available.
MYNAME	should be modified to the name of the local system if UNAME is *not* defined.
ACULAST	is the character required by the ACU as the last character. For most systems, it is a "–".
DATAKIT	should be defined if the system is on a datakit network.
DIALOUT	should be defined if the "C" library routine "dialout" is available.

makefile modification

There are several *make* variable definitions that may need modification:

INSDIR	is the *program* directory (e.g., INSDIR=/usr/lib/uucp). This parameter is used if "make cp" or "make install" is used.
IOCTL	is required to be set if the "ioctl" routine is *not* available in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.
PUBDIR	is a public directory for remote access. This is also the login directory for remote uucp users. It should be the same as that defined in "uucp.h".

SPOOL is the uucp spool directory. This should be the same as that defined in "uucp.h".

XQTDIR is the directory for uuxqt to use for command execution. It is also defined in "uucp.h".

OWNER is the administrative login for uucp.

Compile the system

The command

make install

will make the required directories, compile all programs, set the proper file modes, and copy the programs to the proper directories. This command should be run as *root*. The command

make

will compile the entire system.

The programs *uucp*, *uux*, and *uulog* should be put in "/usr/bin". The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

Files Required For Execution

There are four files that are required for execution. They should reside in the *program* directory. The field separator for all files is a space.

L-devices

This file contains call-unit device and hardwired connection information. The special device files are assumed to be in the */dev* directory. The format for each entry is

type line call-unit speed

where

type is a device type such as ACU or DIR. The field can also be used to specify particular ACUs for some calls by using a suffix on the ACU field, e.g., ACU3. This names should be used in *L.sys*.

line is the device for the line (e.g., cul0).

call-unit is the automatic call unit associated with *line* (e.g., cua0). Hardwired lines have a number "0" in this field.

speed is the line speed.

The line

ACU cul0 cua0 300

would be set up for a system that has device "/dev/cul0" wired to a call-unit "/dev/cua0" for use at 300 baud.

L-dialcodes

This file contains the dialcode abbreviations used in the *L.sys* file (e.g., py, mh, boston). The entry format is

abb dial-seq

where

abb is the abbreviation,

dial-seq is the dial sequence to call that location.

The line

```
py 165-
```

would be set up so that entry py7777 would send 165-7777 to the dial-unit.

USERFILE

This file contains user accessibility information. It specifies four types of constraint:

[1] which files can be accessed by a normal user of the local machine,

[2] which files can be accessed from a remote computer,

[3] which login name is used by a particular remote computer,

[4] whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the format

login,sys [c] path-name [path-name] ...

where

login	is the login name for a user or the remote computer,
sys	is the system name for a remote computer,
c	is the optional *call-back required* flag,
path-name	is a path-name prefix that is acceptable for *sys*.

The constraints are implemented as follows.

[1] When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given on the first line in the *USERFILE* that has the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

[2] When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given on the first line in the file that has the system name that matches the remote machine. If no such line is found, the first one with a *null* system name is used.

[3] When a remote computer logs in, the login name that it uses *must* appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

[4] If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

```
u,m /usr/xyz
```

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

```
dan, /usr/dan
```

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan". (Note that this type restriction is seldom used.)

The lines

```
u,m /usr/xyz /usr/spool
u, /usr/spool
```

allows any remote machine to login with name *u*. If its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool". If it is system *m*, it can send files from paths "/usr/xyz" as well as "/usr/spool".

The lines

```
root, /
, /usr
```

allow any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file. (Note that any file that is to be transferred must be readable by anybody.)

L.sys

Each entry in this file represents one system that can be called by the local uucp programs. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The fields are described below.

system name

The name of the remote system.

time

This is a string that indicates the days-of-week and times-of-day when the system should be called (e.g., MoTuTh0800-1730).

The day portion may be a list containing some of

Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g., 0800-1230). If no time portion is specified, any time of day is assumed to be okay for the call. Note that a time range that spans 0000 is permitted, for example, 0800-0600 means all times are ok other than times between 6 and 8 am.

An optional subfield is available to indicate the minimum time (minutes) before a retry following a failed attempt. The subfield separator is a ",". (e.g., Any,9 means call any time but wait at least 9 minutes after a failure has occurred.)

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g., tty0).

class

This is usually the line speed for the call (e.g., 300). The exception is when the "C" library routine "dialout" is available in which case this is the dialout class.

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation should be one that appears in the *L-dialcodes* file (e.g., mh5900, boston995-9980). For the hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

[expect send] ...

where *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[–send–expect] ...

where the *send* is sent if the prior *expect* is *not* successfully read and the *expect* following the *send* is the next expected string. (e.g., login–login will expect *login*; if it gets it, the program will go on to the next field; if it does not get *login*, it will send *null* followed by a new line, then expect *login* again.)

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.) A number from 1 to 9 may follow the *BREAK* for example, *BREAK1* will send 1 null character instead of the default of 3. Note that *BREAK1* usually works best for 300/1200 baud lines.

A typical entry in the L.sys file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm match all or part of the input string as illustrated in the password field above.

9. Administration

This section indicates some events and files that must be administered for the uucp system. Some administration can be accomplished by *shell files* initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE – sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add the conversation count and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted manually.

TM – temporary data files

These files are created in the *spool* directory while a file is being copied from a remote machine. Their names have the form

TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero. After the entire file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated the file will remain in the spool directory. The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

program/uuclean –pTM

will remove all *TM* files older than three days.

LOG – log entry files

During execution, log information is appended to the *LOGFILE*. If this file is locked by another process, the log information is placed in individual log files which will have prefix *LOG*. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will append the *LOGFILE* with the individual log files. The command

uulog

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically.

The *LOG.* files are created initially with mode 0222. If the program that creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them into the *LOGFILE*.

STST – system status files

These files are created in the spool directory by the *uucico* program. They contain information such as login, dialup or sequence check failures or will contain a *TALKING* status when two machines are conversing. The form of the file name is

STST.*sys*

where *sys* is the remote system name.

For ordinary failures, such as dialup or login, the file will prevent repeated tries for about 55 minutes. This is the default time; it can be changed on an individual system basis by a subfield of the time field in the *L.sys* file. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

LCK – lock files

Lock files are created for each device in use (e.g., automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same device. The form of the lock file name is

LCK..*str*

where *str* is either a device or system name. The files may be left in the spool directory if runs abort (usually only on system crashes). They will be ignored (reused) after 1.5 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

ERRLOG – uucp system error file

This file is created in the *spool* directory to record uucp system errors. Entries in this file should be rare. The messages come from the *ASSERT* statements in the various programs. Wrong modes on files or directories, missing files, and read/write system call failures on the transmission channel may cause entries in the *ERRLOG* file.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but *uucico* will not always be able to execute the request immediately. Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG.* files and started by a crontab entry on an hourly basis. The file could contain the commands

```
/usr/bin/uulog
program/uucico -r1 -sinter
program/uucico -r1
```

The "-r1" option is required to start the *uucico* program in *MASTER* mode. The "-s" option can be used for polling as illustrated in the second line where machine *inter* is being polled. The third line will process all other spooled work.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work that can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program/uuclean -pTM -pC. -pD.
program/uuclean -pST -pLCK -n12
```

can be used. Note that the "-n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "-n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

```
cp spool/LOGFILE  spool/o.LOGFILE
rm spool/LOGFILE
```

can be used.

Login Entry

Two or more logins should be set up for *uucp*. One should be an administrative login: the owner of all the uucp programs, directories and files. All others are used by remote systems to access the uucp system. Each of the "/etc/passwd" entries for the *access* logins should have "*program*/uucico" as the shell to be executed. The login directory should be the public directory (usually /usr/spool/uucppublic). The various *access* login names are used in the *USERFILE* to restrict file access.

File Modes

The programs *uucp*, *uux*, *uucico*, *uulog*, *uuclean* and *uuxqt* should be owned by the *uucp* administrative login with the "setuid" bit set and only execute permissions (e.g., mode 04111). The *L.sys*, *SQFILE* and the *USERFILE*, which are put in the *program* directory should be owned by the *uucp* administrative login and set with mode 0400. The mode of *spool* should be "0755". The mode of *xqtdir* should be "0777". The *L-dialcodes* and the *L-devices* files should have mode 0444.

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

In addition to compilers for C, APL, Pascal, RATFOR, etc., Yacc has also been used for less conventional languages, including a phototypesetter language, several desk calculator languages, a document retrieval system, and a Fortran debugging system.

July 31, 1978

Yacc: Yet Another Compiler-Compiler

Stephen C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

0: Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream. These tokens are organized according to the input structure rules, called *grammar rules*; when one of these rules has been recognized, then user code supplied for this rule, an *action*, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in a portable dialect of C[1] and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

Here, *date*, *month_name*, *day*, and *year* represent structures of interest in the input process; presumably, *month_name*, *day*, and *year* are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

```
July 4, 1776
```

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a *terminal symbol*, while the structure recognized by the parser is called a *nonterminal symbol*. To avoid confusion, terminal symbols will usually be referred to as *tokens*.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

    . . .

month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and *month_name* would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a

month_name was seen; in this case, *month_name* would be a token.

Literal characters such as "," must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is realively easy to add to the above example the rule

```
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development.

The theory underlying Yacc has been described elsewhere.[2,3,4] Yacc has been extensively used in numerous practical applications, including *lint*,[5] the Portable C Compiler,[6] and a system for typesetting mathematics.[7]

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives acknowledgements. Appendix A has a brief example, and Appendix B gives a summary of the Yacc input syntax. Appendix C gives an example using some of the more advanced features of Yacc, and, finally, Appendix D describes mechanisms and syntax no longer actively supported, but provided for historical continuity with older versions of Yacc.

1: Basic Specifications

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the *declarations*, *(grammar)*

rules, and *programs*. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

```
%%
rules
```

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes "'". As in C, the backslash "\" is an escape character within literals, and all the C escapes are recognized. Thus

'\n'	newline
'\r'	return
'\''	single quote "'"
'\\'	backslash "\"
'\t'	tab
'\b'	backspace
'\f'	form feed
'\xxx'	"xxx" in octal

For a number of technical reasons, the NUL character ('\0' or 0) should never be used in grammar rules.

If there are several grammar rules with the same left hand side, the vertical bar "|" can be used to avoid rewriting the left hand side. In addition, the semicolon at the end of a rule can be dropped before a vertical bar. Thus the grammar rules

```
A    :    B C D ;
A    :    E F ;
A    :    G ;
```

can be given to Yacc as

```
A    :    B C D
     |    E F
     |    G
     ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```
empty : ;
```

Names representing tokens must be declared; this is most simply done by writing

```
%token  name1  name2 . . .
```

in the declarations section. (See Sections 3 , 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the *start symbol*, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the *endmarker.* If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it *accepts* the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2: Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{" and "}". For example,

```
A       :       '(' B ')'
                        {       hello( 1, "abc" ); }
```

and

```
XXX     :       YYY ZZZ
                        {       printf("a message\n");
                                flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$$" to some value. For example, an action that does nothing but return the value 1 is

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

```
A     :     B C D ;
```

for example, then $2 has the value returned by C, and $3 the value returned by D.

As a more concrete example, consider the rule

```
expr  :     '(' expr ')' ;
```

The value returned by this rule is usually the value of the *expr* in parentheses. This can be indicated by

```
expr  :     '(' expr ')'          { $$ = $2 ; }
```

By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

```
A     :     B ;
```

frequently need not have an explicit action.

In the examples above, all the actions came at the end of their rules. Sometimes, it is desirable to get control before a rule is fully parsed. Yacc permits an action to be written in the middle of a rule as well as at the end. This rule is assumed to return a value, accessible through the usual mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule

```
A     :     B
                  { $$ = 1; }
            C
                  { x = $2;  y = $3; }
      ;
```

the effect is to set *x* to 1, and *y* to the value returned by C.

Actions that do not terminate a rule are actually handled by Yacc by manufacturing a new nonterminal symbol name, and a new rule matching this name to the empty string. The interior action is the action triggered off by recognizing this added rule. Yacc actually treats the above example as if it had been written:

```
$ACT  :     /* empty */
                  { $$ = 1; }
      ;

A     :     B $ACT C
                  { x = $2;  y = $3; }
      ;
```

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a C function *node*, written so that the call

```
node( L, n1, n2 )
```

creates a node with label L, and descendants n1 and n2, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```
expr    :       expr '+' expr
                        { $$ = node( '+', $1, $3 ); }
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%{" and "%}". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```
%{  int variable = 0;  %}
```

could be placed in the declarations section, making *variable* accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the values are integers: a discussion of values of other types will be found in Section 10.

3: Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by Yacc, or chosen by the user. In either case, the "# define" mechanism of C is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```
yylex(){
        extern int yylval;
        int c;
        ...
        c = getchar();
        ...
        switch( c ) {
                ...
        case '0':
        case '1':
          ...
        case '9':
                yylval = c-'0';
                return( DIGIT );
                ...
                }
        ...
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C or the parser; for example, the use of token names *if* or *while* will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name *error* is reserved for error

handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal *in the declarations section* can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number 0 or negative. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 or negative as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the *Lex* program developed by Mike Lesk.[8] These lexical analyzers are designed to work in close harmony with Yacc parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4: How the Parser Works

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead* token). The *current state* is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

The machine has only four actions available to it, called *shift*, *reduce*, *accept*, and *error*. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex* to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

The *shift* action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

```
IF      shift 34
```

which says, in state 56, if the lookahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The lookahead token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right hand side of a grammar rule, and is prepared to announce that it has seen an instance of the rule, replacing the right hand side by the left hand side. It may be necessary to consult the lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ".") is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

```
.       reduce 18
```

refers to *grammar rule* 18, while the action

```
IF      shift 34
```

refers to *state* 34.

Suppose the rule being reduced is

```
A       :       x y z   ;
```

The reduce action depends on the left hand symbol (A in this case), and the number of symbols on the right hand side (three in this case). To reduce, first pop off the top three states from the stack (In general, the number of states popped equals the number of symbols on the right side of the rule). In effect, these states were the ones put on the stack while recognizing *x*, *y*, and *z*, and no longer serve any useful purpose. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using this uncovered state, and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues. There are significant differences between the processing of the left hand symbol and an ordinary shift of a token, however, so this action is called a *goto* action. In particular, the lookahead token is cleared by a shift, and is not affected by a goto. In any case, the uncovered state contains an entry such as:

```
A       goto 20
```

causing state 20 to be pushed onto the stack, and become the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the states off the stack to go back to the state where the right hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right hand side of the rule is empty, no states are popped off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack, running in parallel with it, holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable *yylval* is copied onto the value stack. After the return from the user code, the reduction is carried out. When the *goto* action is done, the external variable *yyval* is copied onto the value stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The *accept* action indicates that the entire input has been seen and that it matches the specification. This action appears only when the lookahead token is the endmarker, and indicates that the parser has successfully done its job. The *error* action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen, together with the lookahead token, cannot be followed by anything that would result in a legal input. The parser reports an error, and attempts to recover the situation and resume parsing: the error recovery (as opposed to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```
%token DING DONG DELL
%%
rhyme :     sound place
      ;
sound :     DING DONG
      ;
place :     DELL
      ;
```

When Yacc is invoked with the −v option, a file called *y.output* is produced, with a human-readable description of the parser. The *y.output* file corresponding to the above grammar (with some statistics stripped off the end) is:

```
state 0
        $accept : _rhyme $end

        DING shift 3
        . error

        rhyme goto 1
        sound goto 2

state 1
        $accept : rhyme_$end

        $end accept
        . error

state 2
        rhyme : sound_place

        DELL shift 5
        . error

        place goto 4

state 3
        sound : DING_DONG

        DONG shift 6
        . error

state 4
        rhyme : sound place_  (1)

        . reduce 1

state 5
        place : DELL_  (3)

        . reduce 3

state 6
        sound : DING DONG_  (2)

        . reduce 2
```

Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen, and what is yet to come, in each rule. Suppose the input is

DING DONG DELL

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 0. The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token, *DING*, is read, becoming the lookahead token. The action in state 0 on *DING* is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, *DONG*, is read, becoming the lookahead token. The action in state 3 on the token *DONG* is

"shift 6", so state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the lookahead, the parser reduces by rule 2.

```
sound : DING DONG
```

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 0. Consulting the description of state 0, looking for a goto on *sound*,

```
sound  goto 2
```

is obtained; thus state 2 is pushed onto the stack, becoming the current state.

In state 2, the next token, *DELL*, must be read. The action is "shift 5", so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the lookahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right hand side, so one state, 5, is popped off, and state 2 is uncovered. The goto in state 2 on *place*, the left side of rule 3, is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a goto on *rhyme* causing the parser to enter state 1. In state 1, the input is read; the endmarker is obtained, indicated by "$end" in the *y.output* file. The action in state 1 when the endmarker is seen is to accept, successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as *DING DONG DONG*, *DING DONG*, *DING DONG DELL DELL*, etc. A few minutes spend with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5: Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

```
expr  :  expr '-' expr
```

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called *left association*, the second *right association*).

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

```
expr - expr - expr
```

When the parser has read the second expr, the input that it has seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could *reduce* the input by applying this rule; after applying the rule; the input is reduced to *expr*(the left side of the rule). The parser would then read the final part of the input:

```
— expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

```
expr — expr
```

it could defer the immediate application of the rule, and continue reading the input until it had seen

```
expr — expr — expr
```

It could then apply the rule to the rightmost three symbols, reducing them to *expr* and leaving

```
expr — expr
```

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read

```
expr — expr
```

the parser can do two legal things, a shift or a reduction, and has no way of deciding between them. This is called a *shift / reduce conflict.* It may also happen that the parser has a choice of two legal reductions; this is called a *reduce / reduce conflict.* Note that there are never any "Shift/shift" conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing which choice to make in a given situation is called a *disambiguating rule.*

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in favor of shifts. Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the grammar rules, while consistent, require a more complex parser than Yacc can construct. The use of actions within rules can also cause conflicts, if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate, and leads to an incorrect parser. For this reason, Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. For this reason, most previous parser generators have considered conflicts to be fatal errors. Our experience has suggested that this rewriting is somewhat unnatural, and produces slower parsers; thus, Yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an "if-then-else" construction:

```
stat    :    IF '(' cond ')' stat
        |    IF '(' cond ')' stat ELSE stat
        ;
```

In these rules, *IF* and *ELSE* are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be structured according to these rules in two ways:

```
IF ( C1 ) {
        IF ( C2 ) S1
        }
ELSE S2
```

or

```
IF ( C1 ) {
        IF ( C2 ) S1
        ELSE S2
        }
```

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-*ELSE*'d" *IF*. In this example, consider the situation where the parser has seen

```
IF ( C1 ) IF ( C2 ) S1
```

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

```
IF ( C1 ) stat
```

and then read the remaining input,

```
ELSE S2
```

and reduce

```
IF ( C1 ) stat ELSE S2
```

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

can be reduced by the if-else rule to get

```
IF ( C1 ) stat
```

which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things — there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, *ELSE*, and particular inputs already seen, such as

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (−v) option output file. For example, the output corresponding to the above conflict state might be:

23: shift/reduce conflict (shift 45, reduce 18) on ELSE

state 23

```
stat : IF ( cond ) stat_          (18)
stat : IF ( cond ) stat_ELSE stat

ELSE    shift 45
        reduce 18
```

The first line describes the conflict, giving the state and the input symbol. The ordinary state description follows, giving the grammar rules active in the state, and the parser actions. Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

```
IF ( cond ) stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is *ELSE*, it is possible to shift into state 45. State 45 will have, as part of its description, the line

```
stat : IF ( cond ) stat ELSE_stat
```

since the *ELSE* will have been shifted in this state. Back in state 23, the alternative action, described by ".", is to be done if the input symbol is not mentioned explicitly in the above actions; thus, in this case, if the input symbol is not *ELSE*, the parser reduces by grammar rule 18:

```
stat : IF '(' cond ')' stat
```

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to grammar rule numbers. In the *y.output* file, the rule numbers are printed after those rules which can be reduced. In most one states, there will be at most reduce action possible in the state, and this will be the default command. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references[2, 3, 4] might be consulted; the services of a local guru might also be appropriate.

6: Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of *precedence* levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and

construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

```
A .LT. B .LT. C
```

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

```
%right '='
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '=' expr
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       NAME
        ;
```

might be used to structure the input

```
a = b = c*d - e - f*g
```

as follows:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation, but different precedences. An example is unary and binary '-'; unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, %prec, changes the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token name or literal. For example, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%left '+' '-'
%left '*' '/'

%%

expr    :       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '-' expr    %prec '*'
        |       NAME
        ;
```

A token declared by %left, %right, and %nonassoc need not be, but may be, declared by %token as well.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by Yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar; it is a good idea to be sparing with precedences, and use them in an essentially "cookbook" fashion, until some experience has been gained. The *y.output* file is very useful in deciding whether the parser is actually doing what was intended.

7: Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is

legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form

```
stat    :    error
```

would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen. More precisely, the parser will scan ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these; if the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement, and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat    :    error ';'
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input   :    error '\n' { printf( "Reenter last line: " ); } input
                  {       $$ = $4; }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a mechanism that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
yyerrok ;
```

in an action resets the parser to its normal mode. The last example is better written

```
input   :    error '\n'
                  {       yyerrok;
                          printf( "Reenter last line: " );  }
             input
                  {       $$ = $4; }
        ;
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous lookahead token must be cleared. The statement

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after error were to call some

sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old, illegal token must be discarded, and the error state reset. This could be done by a rule like

```
stat    :   error
                    {   resynch();
                        yyerrok ;
                        yyclearin ;  }
        ;
```

These mechanisms are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the error actions required by other portions of the program.

8: The Yacc Environment

When the user inputs a specification to Yacc, the output is a file of C programs, called *y.tab.c* on most systems (due to local file system conventions, the names may differ from installation to installation). The function produced by Yacc is called *yyparse*; it is an integer valued function. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns the value 1, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C program, a program called *main* must be defined, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a library has been provided with default versions of *main* and *yyerror*. The name of this library is system dependent; on many systems the library is accessed by a −ly argument to the loader. To show the triviality of these default programs, the source is given below:

```
main(){
        return( yyparse() );
        }
```

and

```
# include <stdio.h>

yyerror(s) char *s; {
        fprintf( stderr, "%s\n", s );
        }
```

The argument to *yyerror* is a string containing an error message, usually the string "syntax error". The average application will want to do better than this. Ordinarily, the program should keep track of the input line number, and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the lookahead token number at the time the error was detected; this may be of some interest in giving better diagnostics. Since the *main* program is probably supplied by the user (to read arguments, etc.) the Yacc library is useful only in small projects, or in the earliest stages of larger ones.

The external integer variable *yydebug* is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are. Depending on the operating environment, it may be possible to set this variable by using a debugging system.

9: Hints for Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following style hints owe much to Brian Kernighan.

a. Use all capital letters for token names, all lower case letters for nonterminal names. This rule comes under the heading of "knowing who to blame when things go wrong."

b. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.

c. Put all rules with the same left hand side together. Put the left hand side in only once, and let all following rules begin with a vertical bar.

d. Put a semicolon only after the last rule with a given left hand side, and put the semicolon on a separate line. This allows new rules to be easily added.

e. Indent rule bodies by two tab stops, and action bodies by three tab stops.

The example in Appendix A is written following this style, as are the examples in the text of this paper (where space permits). The user must make up his own mind about these stylistic questions; the central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the Yacc parser encourages so called "left recursive" grammar rules: rules of the form

```
name    :       name rest_of_rule ;
```

These rules frequently arise when writing specifications of sequences and lists:

```
list    :       item
        |       list ',' item
        ;
```

and

```
seq     :       item
        |       seq item
        ;
```

In each of these cases, the first rule will be reduced for the first item only, and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as

```
seq     :       item
        |       item seq
        ;
```

the parser would be a bit bigger, and the items would be seen, and reduced, from right to left. More seriously, an internal stack in the parser would be in danger of overflowing if a very long sequence were read. Thus, the user should use left recursion wherever reasonable.

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

```
seq     :       /* empty */
        |       seq item
        ;
```

Once again, the first rule would always be reduced exactly once, before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if Yacc is asked to decide which empty sequence it has seen, when it hasn't seen enough to know!

Lexical Tie-ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings. Or names might be entered into a symbol table in declarations, but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer, and set by actions. For example, suppose a program consists of 0 or more declarations, followed by 0 or more statements. Consider:

```
%{
        int dflag;
%}
  ... other declarations ...

%%

prog    :       decls stats
        ;

decls   :       /* empty */
                        {       dflag = 1; }
        |       decls declaration
        ;

stats   :       /* empty */
                        {       dflag = 0; }
        |       stats statement
        ;

    ... other rules ...
```

The flag *dflag* is now 0 when reading statements, and 1 when reading declarations, *except for the first token in the first statement.* This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This kind of "backdoor" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit the user to use words like "if", which are normally reserved, as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of Yacc; it is difficult to pass information to the lexical analyzer telling it "this instance of 'if' is a keyword, and that instance is a variable". The user can make a stab at it, using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be *reserved*; that is, be forbidden for use as variable names. There are

powerful stylistic reasons for preferring this, anyway.

10: Advanced Topics

This section discusses a number of advanced features of Yacc.

Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros YYACCEPT and YYERROR. YYACCEPT causes *yyparse* to return the value 0; YYERROR causes the parser to behave as if the current input symbol had been a syntax error; *yyerror* is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple endmarkers or context-sensitive syntax checking.

Accessing Values in Enclosing Rules.

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit, but in this case the digit may be 0 or negative. Consider

```
sent    :       adj noun verb adj noun
                        { look at the sentence . . . }
        ;

adj     :       THE             {       $$ = THE; }
        |       YOUNG           {       $$ = YOUNG; }
        . . .
        ;

noun    :       DOG
                        {       $$ = DOG; }
        |       CRONE
                        {       if( $0 == YOUNG ){
                                        printf( "what?\n" );
                                        }
                                $$ = CRONE;
                                }
        ;
        . . .
```

In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG. Obviously, this is only possible when a great deal is known about what might precede the symbol *noun* in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism will save a great deal of trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers. Yacc can also support values of other types, including structures. In addition, Yacc keeps track of the types, and inserts appropriate union member names so that the resulting parser will be strictly type checked. The Yacc value stack (see Section 4) is declared to be a *union* of the various types of values desired. The user declares the union, and associates union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, Yacc will automatically insert the appropriate union name, so that no unwanted conversions will take place. In addition, type checking commands such as *Lint*[5] will be far more silent.

There are three mechanisms used to provide for this typing. First, there is a way of defining the union; this must be done by the user since other programs, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where Yacc can not easily determine the type.

To declare the union, the user includes in the declaration section:

```
%union {
        body of union ...
        }
```

This declares the Yacc value stack, and the external variables *yylval* and *yyval*, to have type equal to this union. If Yacc was invoked with the **-d** option, the union declaration is copied onto the *y.tab.h* file. Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. Thus, the header file might also have said:

```
typedef union {
        body of union ...
        } YYSTYPE;
```

The header file must be included in the declarations section, by use of %{ and %}.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction

```
< name >
```

is used to indicate a union member name. If this follows one of the keywords %token, %left, %right, and %nonassoc, the union member name is associated with the tokens listed. Thus, saying

```
%left <optype> '+' '-'
```

will cause any reference to values returned by these two tokens to be tagged with the union member name *optype*. Another keyword, %type, is used similarly to associate union member names with nonterminals. Thus, one might say

```
%type <nodetype> expr stat
```

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as $0 – see the previous subsection) leaves Yacc with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name, between < and >, immediately after the first $. An example of this usage is

```
rule    :       aaa { $<intval>$ = 3; } bbb
                        { fun( $<intval>2, $<other>0 ); }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix C. The facilities in this subsection are not triggered until they are used: in particular, the use of %type will turn on these mechanisms. When they are used, there is a fairly strict level of checking. For example, use of $n or $$ to refer to something with no defined type is diagnosed. If these facilities are not triggered, the Yacc value stack is used to hold *int*'s, as was true historically.

11: Acknowledgements

Yacc owes much to a most stimulating collection of users, who have goaded me beyond my inclination, and frequently beyond my ability, in their endless search for "one more feature". Their irritating unwillingness to learn how to do things my way has usually led to my doing things their way; most of the time, they have been right. B. W. Kernighan, P. J. Plauger, S. I. Feldman, C. Imagna, M. E. Lesk, and A. Snyder will recognize some of their ideas in the current version of Yacc. C. B. Haley contributed to the error recovery algorithm. D. M. Ritchie, B. W. Kernighan, and M. O. Harris helped translate this document into English. Al Aho also deserves special credit for bringing the mountain to Mohammed, and other favors.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. A. V. Aho and S. C. Johnson, "LR Parsing," *Comp. Surveys* 6(2) pp. 99-124 (June 1974).
3. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic Parsing of Ambiguous Grammars," *Comm. Assoc. Comp. Mach.* 18(8) pp. 441-452 (August 1975).
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
5. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (December 1977).
6. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages*, (January 1978).
7. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Comm. Assoc. Comp. Mach.* 18 pp. 151-157 (March 1975).
8. M. E. Lesk, "Lex — A Lexical Analyzer Generator," Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Appendix A: A Simple Example

This example gives the complete Yacc specification for a small desk calculator; the desk calculator has 26 registers, labeled "a" through "z", and accepts arithmetic expressions made up of the operators +, −, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a Yacc specification, the desk calculator does a reasonable job of showing how precedences and ambiguities are used, and demonstrating simple error recovery. The major oversimplifications are that the lexical analysis phase is much simpler than for most applications, and the output is produced immediately, line by line. Note the way that decimal and octal integers are read in by the grammar rules; This job is probably better done by the lexical analyzer.

```
%{
# include <stdio.h>
# include <ctype.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS    /* supplies precedence for unary minus */

%%    /* beginning of rules section */

list    :       /* empty */
        |       list stat '\n'
        |       list error '\n'
                        {       yyerrok; }
        ;

stat    :       expr
                        {       printf( "%d\n", $1 ); }
        |       LETTER '=' expr
                        {       regs[$1] = $3; }
        ;

expr    :       '(' expr ')'
                        {       $$ = $2; }
        |       expr '+' expr
                        {       $$ = $1 + $3; }
        |       expr '-' expr
                        {       $$ = $1 - $3; }
```

```
        |       expr '*' expr
                        {       $$ = $1 * $3; }
        |       expr '/' expr
                        {       $$ = $1 / $3; }
        |       expr '%' expr
                        {       $$ = $1 % $3; }
        |       expr '&' expr
                        {       $$ = $1 & $3; }
        |       expr '|' expr
                        {       $$ = $1 | $3; }
        |       '-' expr        %prec UMINUS
                        {       $$ = - $2; }
        |       LETTER
                        {       $$ = regs[$1]; }
        |       number
        ;

number:         DIGIT
                        {       $$ = $1;   base = ($1==0) ? 8 : 10; }
        |       number DIGIT
                        {       $$ = base * $1 + $2; }
        ;

%%      /* start of programs */

yylex() {               /* lexical analysis routine */
        /* returns LETTER for a lower case letter, yylval = 0 through 25 */
        /* return DIGIT for a digit, yylval = 0 through 9 */
        /* all other characters are returned immediately */

        int c;

        while( (c=getchar()) == ' ' ) {/* skip blanks */ }

        /* c is now nonblank */

        if( islower( c ) ) {
                yylval = c - 'a';
                return ( LETTER );
                }
        if( isdigit( c ) ) {
                yylval = c - '0';
                return( DIGIT );
                }
        return( c );
        }
```

Appendix B: Yacc Input Syntax

This Appendix has a description of the Yacc input syntax, as a Yacc specification. Context dependencies, etc., are not considered. Ironically, the Yacc input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule, immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier, and decide whether the next token (skipping blanks, newlines, comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERS, but never as part of C_IDENTIFIERs.

```
        /* grammar for the input to Yacc */

        /* basic entities */
%token  IDENTIFIER      /*  includes identifiers  and literals */
%token  C_IDENTIFIER    /*  identifier (but not literal) followed by colon   */
%token  NUMBER                  /*   [0-9]+   */

        /* reserved words:  %type => TYPE, %left => LEFT, etc. */

%token  LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token  MARK  /* the %% mark */
%token  LCURL /* the %{ mark */
%token  RCURL /* the %} mark */

        /* ascii character literals stand for themselves */

%start  spec

%%

spec    :       defs MARK rules tail
        ;

tail    :       MARK { In this action, eat up the rest of the file }
        |       /* empty: the second MARK is optional */
        ;

defs    :       /* empty */
        |       defs def
        ;

def     :       START IDENTIFIER
        |       UNION { Copy union definition to output }
        |       LCURL { Copy C code to output file } RCURL
        |       ndefs rword tag nlist
        ;

rword   :       TOKEN
        |       LEFT
        |       RIGHT
```

```
        |       NONASSOC
        |       TYPE
        ;

tag     :       /* empty: union tag is optional */
        |       '<' IDENTIFIER '>'
        ;

nlist   :       nmno
        |       nlist nmno
        |       nlist ',' nmno
        ;

nmno    :       IDENTIFIER              /* NOTE: literal illegal with %type */
        |       IDENTIFIER NUMBER       /* NOTE: illegal with %type */
        ;

        /* rules section */

rules   :       C_IDENTIFIER rbody prec
        |       rules rule
        ;

rule    :       C_IDENTIFIER rbody prec
        |       '|' rbody prec
        ;

rbody   :       /* empty */
        |       rbody IDENTIFIER
        |       rbody act
        ;

act     :       '{' { Copy action, translate $$, etc. } '}'
        ;

prec    :       /* empty */
        |       PREC IDENTIFIER
        |       PREC IDENTIFIER act
        |       prec ';'
        ;
```

Appendix C: An Advanced Example

This Appendix gives an example of a grammar using some of the advanced features discussed in Section 10. The desk calculator example in Appendix A is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants, the arithmetic operations +, −, *, /, unary −, and = (assignment), and has 26 floating point variables, "a" through "z". Moreover, it also understands *intervals*, written

(x , y)

where x is less than or equal to y. There are 26 interval valued variables "A" through "Z" that may also be used. The usage is similar to that in Appendix A; assignments return no value, and print nothing, while expressions print the (floating or interval) value.

This example explores a number of interesting features of Yacc and C. Intervals are represented by a structure, consisting of the left and right endpoint values, stored as *double*'s. This structure is given a type name, INTERVAL, by using *typedef*. The Yacc value stack can also contain floating point scalars, and integers (used to index into the arrays holding the variable values). Notice that this entire strategy depends strongly on being able to assign structures and unions in C. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions: division by an interval containing 0, and an interval presented in the wrong order. In effect, the error recovery mechanism of Yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (e.g. scalar or interval) of intermediate expressions. Note that a scalar can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through Yacc: 18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the two input lines:

2.5 + (3.5 − 4.)

and

2.5 + (3.5 , 4.)

Notice that the 2.5 is to be used in an interval valued expression in the second example, but this fact is not known until the "," is read; by this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator: one when the left operand is a scalar, and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflicts will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive, but not very general. If there were many kinds of expression types, instead of just two, the number of rules needed would increase dramatically, and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value, and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C library routine *atof* is used to do the actual conversion from a character string to a double precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser, and thence error recovery.

```
%{

# include <stdio.h>
# include <ctype.h>

typedef struct interval {
        double lo, hi;
        } INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[ 26 ];
INTERVAL vreg[ 26 ];

%}

%start  lines

%union  {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG         /* indices into dreg, vreg arrays */

%token <dval> CONST             /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */

        /* precedence information about the operators */

%left  '+' '-'
%left  '*' '/'
%left  UMINUS     /* precedence for unary minus */

%%

lines   :       /* empty */
        |       lines line
        ;

line    :       dexp '\n'
                        {       printf( "%15.8f\n", $1 ); }
        |       vexp '\n'
                        {       printf( "(%15.8f , %15.8f )\n", $1.lo, $1.hi ); }
        |       DREG '=' dexp '\n'
                        {       dreg[$1] = $3; }
        |       VREG '=' vexp '\n'
```

```
                { vreg[$1] = $3; }
        |   error '\n'
                { yyerrok; }
        ;

dexp    :   CONST
        |   DREG
                { $$ = dreg[$1]; }
        |   dexp '+' dexp
                { $$ = $1 + $3; }
        |   dexp '-' dexp
                { $$ = $1 - $3; }
        |   dexp '*' dexp
                { $$ = $1 * $3; }
        |   dexp '/' dexp
                { $$ = $1 / $3; }
        |   '-' dexp    %prec UMINUS
                { $$ = -$2; }
        |   '(' dexp ')'
                { $$ = $2; }
        ;

vexp    :   dexp
                { $$.hi = $$.lo = $1; }
        |   '(' dexp ',' dexp ')'
                {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi ){
                        printf( "interval out of order\n" );
                        YYERROR;
                        }
                }
        |   VREG
                { $$ = vreg[$1]; }
        |   vexp '+' vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo; }
        |   dexp '+' vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo; }
        |   vexp '-' vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi; }
        |   dexp '-' vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi; }
        |   vexp '*' vexp
                { $$ = vmul( $1.lo, $1.hi, $3 ); }
        |   dexp '*' vexp
                { $$ = vmul( $1, $1, $3 ); }
        |   vexp '/' vexp
                { if( dcheck( $3 ) ) YYERROR;
                  $$ = vdiv( $1.lo, $1.hi, $3 ); }
```

```
        |       dexp '/' vexp
                        {       if( dcheck( $3 ) ) YYERROR;
                                $$ = vdiv( $1, $1, $3 ); }
        |       '-' vexp        %prec UMINUS
                        {       $$.hi = -$2.lo;  $$.lo = -$2.hi;  }
        |       '(' vexp ')'
                        {       $$ = $2; }
        ;

%%

# define BSZ 50     /* buffer size for floating point numbers */

        /* lexical analysis */

yylex(){
        register c;

        while( (c=getchar()) == ' ' ){ /* skip over blanks */ }

        if( isupper( c ) ){
                yylval.ival = c - 'A';
                return( VREG );
                }
        if( islower( c ) ){
                yylval.ival = c - 'a';
                return( DREG );
                }

        if( isdigit( c ) || c=='.' ){
                /* gobble up digits, points, exponents */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ; (cp-buf)<BSZ ; ++cp,c=getchar() ){

                        *cp = c;
                        if( isdigit( c ) ) continue;
                        if( c == '.' ){
                                if( dot++ || exp ) return( '.' );  /* will cause syntax error */
                                continue;
                                }

                        if( c == 'e' ){
                                if( exp++ ) return( 'e' );  /* will cause syntax error */
                                continue;
                                }

                        /* end of number */
                        break;
                        }
                *cp = '\0';
                if( (cp-buf) >= BSZ ) printf( "constant too long: truncated\n" );
```

```
            else ungetc( c, stdin );  /* push back last char read */
            yylval.dval = atof( buf );
            return( CONST );
            }
        return( c );
        }

INTERVAL hilo( a, b, c, d ) double a, b, c, d; {
        /* returns the smallest interval containing a, b, c, and d */
        /* used by *, / routines */
        INTERVAL v;

        if( a>b ) { v.hi = a;   v.lo = b; }
        else { v.hi = b;   v.lo = a; }

        if( c>d ) {
            if( c>v.hi ) v.hi = c;
            if( d<v.lo ) v.lo = d;
            }
        else {
            if( d>v.hi ) v.hi = d;
            if( c<v.lo ) v.lo = c;
            }
        return( v );
        }

INTERVAL vmul( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );
        }

dcheck( v ) INTERVAL v; {
        if( v.hi >= 0. && v.lo <= 0. ){
            printf( "divisor interval contains 0.\n" );
            return( 1 );
            }
        return( 0 );
        }

INTERVAL vdiv( a, b, v ) double a, b;  INTERVAL v; {
        return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );
        }
```

Appendix D: Old Features Supported but not Encouraged

This Appendix mentions synonyms and features which are supported for historical continuity, but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes "'".
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or _, the type number of the literal is defined, just as if the literal did not have the quotes around it. Otherwise, it is difficult to find the value for such literals.

   The use of multi-character literals is likely to mislead those unfamiliar with Yacc, since it suggests that Yacc is doing a job which must be actually done by the lexical analyzer.
3. Most places where % is legal, backslash "\" may be used. In particular, \\ is the same as %%, \left the same as %left, etc.
4. There are a number of other synonyms:

   %< is the same as %left
   %> is the same as %right
   %binary and %2 are the same as %nonassoc
   %0 and %term are the same as %token
   %= is the same as %prec

5. Actions may also have the form

   ={ . . . }

   and the curly braces can be dropped if the action is a single C statement.
6. C code between %{ and %} used to be permitted at the head of the rules section, as well as in the declaration section.

Plexus Release 1.0 of the UNIX Virtual Protocol Machine

This document draws heavily from the memorandum on VPM supplied in the Programmer's Manual for UNIX System III, Volume 2B (October 1981).

Changes have been made to reflect the Plexus implementation of VPM.

ABSTRACT

This document describes the initial release of the Virtual Protocol Machine (VPM), a UNIX* synchronous communication subsystem, as implemented on a Plexus computer system. The Plexus release of VPM is built around the Intelligent Communications Processor (ICP), a Z8000 based microcomputer that connects to the MULTIBUS of a Plexus computer. The VPM is a software construct for implementing link protocols on the ICP in a high-level language.

A compiler, vpmc, is provided to translate a high-level description of a protocol (protocol script) into the instruction set of the virtual machine. Vpmc supports C-like control-flow constructs, a modest subset of C-like statements and expressions, and a set of communication primitives that permit implementation of byte-oriented protocols such as BISYNC. (Primitives that support bit-oriented protocols such as HDLC have been defined and will be available in a later release of VPM.) An interpreter is provided that runs in the ICP and interprets the virtual machine instruction set. A UNIX driver provides the interface between the user process's open, close, read, and write calls and the protocol script being executed by the interpreter. Besides providing the benefits of a high-level language implementaton of protocols, the VPM approach permits portable protocol implementations.

The VPM software consists of five components:

1. vpmc: a UNIX compiler for the protocol description language.

2. VPM interpreter: the ICP program that controls the overall operation of the ICP and interprets the protocol script.

---

* UNIX is a trademark of Bell Laboratories.

3. si.c: the UNIX driver that provides the interface to the VPM. This source is not available for distribution.

4. vpmstart: a UNIX command that copies a load module into the ICP and starts it.

5. vpmtrace: a UNIX command that prints an event trace for debugging while the protocol is running.

The procedures for installation and use of the VPM commands and the VPM driver are described; the pertinent manual entries are attached.

INTRODUCTION

The Virtual Protocol Machine (VPM) is a UNIX synchronous communications subsystem built around the ICP microcomputer.

The VPM is a software construct for implementing link protocols on the ICP using a high-level language. A compiler, vpmc, is provided to translate a high-level description of a protocol (protocol script) into the instruction set of the virtual machine. Vpmc uses a variant of Ratfor[1] as a front end to provide control-flow constructs such as if-else, for, while, switch, and repeat-until, as well as other benefits. Vpmc supports a modest subset of C-like statements and expressions, plus a set of communications primitives that permit succinct and easily-understood implementations of byte-oriented protocols such as BISYNC. These primitives allow the protocol scripts to reflect the essential structure of the protocol, while hiding details that arise from a particular hardware-software environment. (Primitives that support bit-oriented protocols such as HDLC have been defined and will be available in a later release of VPM.) An interpreter is provided that runs in the ICP and interprets the virtual machine instruction set. This program also controls the communications line(s) and provides the interface to the UNIX host machine. The compiled protocol script is loaded with the interpreter into the ICP. A UNIX driver provides the interface between the user process's open, close, read, and write calls and the protocol script executed by the interpreter in the ICP.

Besides providing the benefits of a high-level language implementations of protocols, such as ease of programming and maintainability, the VPM approach permits portable protocol implementations. Portability can be achieved in several ways. First, because the interpreter and the compiled protocol script execute in the ICP, they are the same regardless of the

---

1. B. W. Kernighan, RATFOR - A Preprocessor for a Rational Fortran, Bell Laboratories.

software running in the main CPU or, for that matter, regardless of the CPU itself. More general forms of portability are also possible. The instruction set of the virtual machine can be translated into almost any assembly language using one of the UNIX macro processors, such as m4[2]. This does not require that the assembler for the target machine have a macro expansion capability. Another possibility for portability because Ratfor is used as a front-end: by limiting a protocol script to a statement and expression syntax acceptable to a Fortran compiler, the protocol is portable to machines that support Fortran in a suitable real-time environment. Finally, minor changes to a protocol script will yield a C implementation of the protocol. With any of these methods, the functions provided by the primitives (including the interfacing with communication devices and the execution environment) must be supplied by suitable library routines or system calls.

PLEXUS RELEASE 1.0

Plexus release 1.0 of VPM is restricted to byte-oriented half-duplex protocols such as BISYNC.

This release of the VPM software is intended for use with UNIX Sys3. Operation with other versions of UNIX has not been tested. The VPM software consists of five components:

1. vpmc: a UNIX compiler for the protocol description language.
2. VPM interpreter: the ICP program that controls the overall operation of the ICP and interprets the protocol script.
3. si.c: the UNIX driver that provides the interface to the VPM. This source is not available for distribution.
4. vpmstart: a UNIX command that copies a load module into the ICP and starts it.
5. vpmtrace: a UNIX command that prints an event trace for debugging while the protocol is running.

A release tape containing the VPM software is available from Plexus Computers. Installation procedures are described in the appendix to this document.

---

2. B. W. Kernighan, The M4 Macro Processor, Bell Laboratories.

APPENDIX

This appendix has two parts: the first gives configuration guidelines for VPM and the TTY ICPs and tells how to install and boot VPM; the second gives procedures for compiling and link-loading protocol scripts.

1. Installing and Booting VPM

Each TTY ICP can support up to eight users. If VPM is also installed, an additional dedicated ICP is required for the VPM. Therefore, a P/40 with 32 users and a VPM requires FIVE ICPs. A P/25 with 8 users and a VPM requires TWO ICPs.

For both P/40s and P/25s, the lowest numbered ICP must be the VPM. Thus while VPM is operating, ports 0-7 may not be used as TTY ports; users' TTY ports must be numbered beginning with 8. For example, on a P/40 with 16 users and VPM, the VPM uses ports 0-7 and users' TTYs are numbered 8-23. The port assignments are changed by modifying the file /etc/inittab for use with VPM.

If you want VPM in operation only intermittently, the VPM ICP can do double duty as a TTY ICP. Seven out of eight devices remain available on the VPM ICP even when VPM is not operating. The one exception is the wirewrapped port (item 4 in section 1.1 below). You can switch back and forth by alternating between versions of /etc/rc that call different versions of /etc/inittab.

The VPM software is part of Sys3 Release 1.1 and is loaded onto your system during the ordinary installation of Release 1.1. See the Plexus Sys3 1.1 Release Notice for more information about loading Sys3 Release 1.1. After 1.1 is installed, five basic steps are required to bring up VPM. The following sections describe each step.

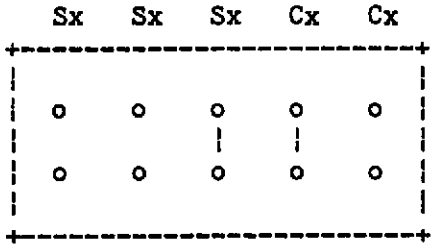
1. Perform several small hardware changes.
2. Create the VPM devices.
3. Change the modes of /dev/ic files.
4. Modify /etc/inittab.
5. Boot Sys3 with a new VPM kernel.

1.1 Hardware Installation

The ICP(s) that are to be the VPM(s) require the following special hardware features:

1. The line(s) that are to be synchronous require the Carrier Detect signal and Clear to Send signal to be strapped on the ICP. (See Plexus P/40 User's Manual.)

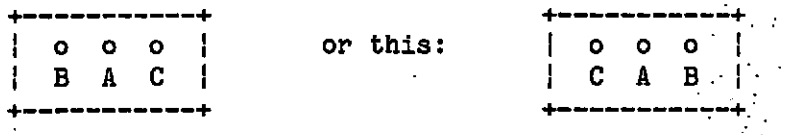
2. The pin-pairs 3 and 4 must be jumpered for synchronous transmission. Set up the 10-pin jumper network to look like the diagram below.



3. If necessary, update the VPM ICP to hardware level W.

4. Each port using synchronous transmission must be configured for external clock. This is accomplished via a pair of three-pin jumper networks for each port on that ICP. The jumper networks are designated Tx and Xx, where `x' means the port number.

   All Tx and Xx jumper networks look like this:



   Normally, the center pin (A) is jumpered to either outer pin (B or C) using a two-pin female jumper block. However, configuring a port for external clocking requires wirewrapping pin B to pin C on both jumper networks for the port. (Wirewrapping is necessary because pin B and pin C are never contiguous.)

   For example, to strap port 0 for external clock, do the following:

   a. Remove the two-pin jumper blocks from T0 and from X0.

   b. Using a wirewrap tool, connect T0-B to T0-C.

   c. Using a wirewrap tool, connect X0-B to X0-C.

5. The connection cable between the VPM ICP port and the synchronous modem is a specially strapped Plexus RS232C modem cable. The

following RS232C modem cable leads must be strapped:

VPM ICP Port RS232 Leads	to	Modem Cable RS232 Leads
2		3
3		2
4		5
5		4
6		20
7		7
8		8
15		15
17		17
20		6

1.2 Create the VPM Devices

Login as root and bring your system to init state 1. Then use **mknod**(1M) to create a node for each VPM line and each ICP:

```
/etc/mknod /dev/vpm? c <major> <minor>
```

where major and minor are both octal numbers. Major is determined by vpm's position in the cdevsw table. Minor is defined as follows: the two most significant bits denote the physical ICP number (0-3), the next two bits denote the VPM protocol number (0-3), and the four least significant bits denote the physical line number on the ICP. For example, if ICPs 0 and 1 are to be used for VPM using protocol number 1 and line number 3, then the minor device numbers should be 023 and 0123, respectively.

The default major device number for VPM in Plexus Sys3 Release 1.1 is 18. Therefore, the mknod step might proceed as follows:

```
mknod /dev/vpm0 c 18 0
mknod /dev/vpm1 c 18 1
mknod /dev/vpm2 c 18 2
mknod /dev/vpm3 c 18 3
```

1.3 Change Modes of /dev/ic Files

Add read and wrote modes to the /dev/ic files as follows

```
chmod +rw /dev/ic/*
```

1.4 Modify /etc/inittab

Change the logical device assignments in /etc/inittab so that only VPM devices (/dev/vpm0 - /dev/vpm7) are assigned ports 0-7 on ICP0. TTY devices formerly assigned these ports should receive port assignments on different ICPs.

1.5 Boot the VPM Kernel

Four copies of VPM kernels are included in Sys3 Release 1.1. They are as follows:

Kernel	Disk Controller	Number of Users
/sys3vim.16	IMSC	8 or 16
/sys3vim.32	IMSC	32
/sys3vis.16	iSBC	8 or 16
/sys3vis.32	iSBC	32

Login as root and bring your system to init state 1. Before booting the VPM kernel, back up the regular kernel, /sys3, and copy the appropriate VPM kernel to the file /sys3 as follows:

```
cp /sys3 /OLDsys3
cp <VPM kernel from list above> /sys3
```

This is necessary for the ps(1) command to work properly.

Then shutdown and reboot normally, following the procedures in the Plexus User's Manual.

2. Compiling and Loading VPM Scripts

This section gives the steps to compile and load VPM scripts.

1. You must be in the directory /usr/src/cmd/vpm, so issue the command

   ```
   cd /usr/src/cmd/vpm
   ```

2. Then move your protocol script into this directory, renaming it vpmscript.r.

   ```
   mv <your protocol script name> vpmscript.r
   ```

3. Execute the following command

```
make -f vpmscript.mk
```

This compiles the script in **vpmscript.r** and link-loads this compiled script with the rest of the VPM ICP kernel. The object file created in this step is called **vpm0**; it is down-loadable into the VPM ICP.

4. To download **vpm0**, use either of the programs **dnld**(1) or **vpmstart**(1).

5. You may combine up to four scripts in one download module. Scripts to be combined in this way must be called **proto?code.s**, where ``?'' represents a number from 0 to 3. To combine scripts, you must modify the file **vpmscript.mk**. inserting instructions for each script (**proto?code.s**) to be compiled into a **proto?.s** file, where ``?'' represents a number from 0 to 3. The following lines accomplish this compilation; note that this whole series of steps must be done for each script. Therefore, you must copy these lines into the file **vpmscript.mk** once for each script, making sure you make the appropriate substitution of a number for the ``?''.

```
(1) vpmc -m -x -o vpmscript <source script>

(2) fgrep define sas_tempc > sas_define

(3) cat /usr/include/icp/opdef.h sas_tempc | /lib/cpp > tf

(4) /usr/lib/vpm/vratfor < tf > tg

(5) cp tg /usr/src/uts/z8000/icp/vpmicp/proto?code.s

(6) cat sas_define proto?.s > th

(7) cp th /usr/src/uts/z8000/icp/vpmicp/proto?.s
```

Bell Laboratories

subject: **Release 2.0 of the UNIX Virtual Protocol Machine**

date: **November 21, 1980**

from: **P. F. Long**

C. Mee III

ABSTRACT

This memorandum describes the second release of the UNIX* Virtual Protocol Machine (VPM). VPM is a general-purpose synchronous UNIX communications interface that allows link-level protocols such as BISYNC and HDLC to be implemented on the KMC11-B (a DEC microcomputer) in a high-level language. The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs.

The first release of VPM supports a class of byte-oriented half-duplex protocols collectively known as BISYNC. The present release adds support for bit-oriented, full-duplex protocols such as the international standard High-Level Data Link Control (HDLC). Other features of Release 2.0 include:

1. an increase in the number of buffers which the interpreter can accept at one time;

2. additional debugging facilities;

3. provisions for interprocess communication between the protocol script and a UNIX driver or a user process; and

4. a cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

The procedures for adding VPM Release 2.0 to a UNIX 3.0 system and testing it to ensure proper operation are given. Manual entries for VPM are attached.

---

* UNIX is a trademark of Bell Laboratories.

MEMORANDUM FOR FILE

Introduction

This memorandum describes the second release of the UNIX* Virtual Protocol Machine (VPM). The first release was described in a previous memorandum [1], which should be read as background for this memorandum. See also the attached manual entry for *vpm*(4).

VPM is a general-purpose UNIX interface for synchronous communications lines. VPM allows link-level protocols such as BISYNC and HDLC to be implemented on the DEC KMC11-B microcomputer in a high-level language. The hardware required to support VPM is a PDP-11/70, /45, or /34, or a VAX-11/780 host computer, a KMC11-B microcomputer, and a DMC11-DA, -FA, or -FD synchronous communications interface. All of the above items are manufactured by DEC. The use of the KMC microcomputer allows the VPM to perform direct-memory-access (DMA) transfers to and from main memory. The link-level communications protocol is executed by the VPM interpreter running in the KMC microcomputer. This implementation technique leads to a portable protocol representation and efficient protocol execution.

The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the KMC, and several utility programs. The compiler, which executes in the host computer, translates a protocol described in a high-level language into a load module for the KMC. The load module contains the VPM interpreter and a compiled representation of the protocol. The interpreter executes the protocol, communicates with the UNIX driver in the host computer, and controls the communications line interface.

The first release of VPM supported a large class of protocols collectively known as BISYNC. These protocols are distinguished by the use of control characters to provide framing and transparency. At the frame level, these protocols operate in a half-duplex manner, although they sometimes use full-duplex communications facilities to reduce the time required to reverse the direction of transmission.

Release 2.0 of VPM adds support for bit-oriented, full-duplex protocols. This class of protocols includes IBM's Synchronous Data Link Control (SDLC) and the international standard High-Level Data Link Control (HDLC). LAPB, a subset of HDLC which is the link-level protocol specified in the BX.25 Bell System Standard, has been implemented using VPM and is available with this release [2,3]. The interpreter used for bit-oriented protocols is different from that used for character-oriented (BISYNC) protocols. The appropriate interpreter is selected by means of a compiler option.

Other features of Release 2.0 include:

1. an increase in the number of transmit and receive buffers which the interpreter can accept at one time,
2. additional debugging facilities,
3. provisions for interprocess communication between the protocol script and a UNIX driver or a user process, and
4. a cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

---

* UNIX is a trademark of Bell Laboratories.

Support for Bit-Oriented Protocols

The capability to use bit-oriented protocols such as HDLC is provided by a new set of communications primitives. These primitives are frame-oriented and non-blocking, whereas the BISYNC primitives are character-oriented and blocking. The new primitives are fully described in the attached manual entry for *vpmc*(1C). An overview of these primitives follows.

The VPM interpreter maintains a set of queues for transmit buffers and another set of queues for receive buffers. When a transmit buffer is passed to the KMC by the UNIX driver, the buffer is appended to the unopened-transmit-buffer queue. The protocol script in the KMC obtains a transmit buffer from the unopened-transmit-buffer queue by means of the *getxfrm* primitive; the buffer is then said to be *open*. In order to get (open) a transmit buffer, the script must provide a transmit-sequence number. This sequence number must be in the range 0-7 and must be distinct from the sequence number currently assigned to every other currently-open transmit buffer. This sequence number is used to identify the buffer for subsequent calls to the *xmtfrm* and *rtnxfrm* primitives. The *xmtfrm* primitive initiates transmission of the specified buffer, using the control information specified by a previous *setctl* primitive. Transmission proceeds asynchronously. The script can test for completion of an output transfer by means of the *xmtbusy* primitive. Open transmit buffers can be transmitted any number of times. When the script decides that a buffer has successfully been received at the destination, it notifies the interpreter by means of the *rtnxfrm* primitive. This causes the buffer to be placed on the transmit-buffer-return queue; the buffer is then no longer considered to be open and the sequence number can be reused. The driver is notified as soon as possible that the buffer has been closed. The buffer is then removed from the transmit-buffer-return queue.

When a receive buffer is passed to the KMC by the driver, the buffer is placed on the empty-receive-buffer queue. When the first byte of a new frame arrives, an empty receive buffer is obtained from the empty-receive-buffer queue and the incoming characters are placed into the buffer as they arrive. An incoming frame will be discarded if the frame is too short (less than four bytes including CRC), if the frame is too long to fit in the receive buffer, or if the CRC is incorrect. If a frame is received successfully, the buffer is placed on the completed-receive-frame queue, otherwise the buffer is returned to the empty-receive-buffer queue. When the script executes a *rcvfrm* primitive, the buffer at the head of the completed-receive-frame queue is removed from that queue and becomes the current receive buffer. If the script subsequently executes a *rtnrfrm* primitive before executing another *rcvfrm* primitive, the current receive buffer is placed on the receive-buffer-return queue. If the script executes a *rcvfrm* primitive before executing a *rtnrfrm* primitive, the current receive buffer, if any, is returned to the empty-receive-frame queue. Buffers on the receive-buffer-return queue are returned to the driver at the first opportunity.

If the empty-receive-buffer queue is empty when the first byte of a new frame is received, the first five bytes of the frame are retained in a staging area and the remainder of the frame is discarded. This allows a protocol script to receive a control frame (up to seven bytes including CRC) when no data buffer is available. When the next *rcvfrm* primitive is executed, the script will receive the information in the staging area along with an indication that the remainder of the frame has been discarded. If another frame arrives while the staging area is thus occupied, the new frame is discarded entirely.

A count is kept of the number of frames discarded for each reason. These counters may be read and reset from the host computer.

The VPM Split Driver

Since the VPM interpreter and a protocol script generally use most of the memory of the KMC, any higher levels of protocol that are required must be executed by the host CPU. The purpose of the VPM split driver is to provide a framework in which higher-level protocols can be implemented conveniently using low-level routines in the VPM driver to communicate with the

interpreter in the KMC.

A set of functions has been written which provides a general-purpose interface to the link-level protocol being executed by the interpreter in the KMC. Their capabilities include a means to queue transmit and empty receive buffers for use by the protocol script in the KMC, to start and stop the script, and to send commands to and receive reports from the script. A means of getting a copy of and resetting the VPM interpreter's error counters is also provided. These functions will be referred to as interface functions or collectively as the interface module. Appendix 1 contains a description of each of these routines.

To implement higher levels of a protocol as a UNIX device driver, a set of routines must be written to implement the standard UNIX system calls: *open*, *close*, *read*, *write*, and *ioctl* as well as the required protocol. These routines will be referred to as protocol functions or collectively as a protocol module. The standard VPM driver does not implement a higher-level protocol but instead provides a transparent user interface that can be used by applications that supply their own higher levels of protocol. This driver can be used as an example for those interested in writing a different protocol module. Appendix 2 contains a description of these routines.

At least two other protocol modules have been written thus far. They are the Synchronous Terminal Interface [4,*st*(4)], and the BANCS THP Interface.

Release 2.0 of VPM allows up to four different VPM protocol modules to be executing simultaneously. One KMC and one interface-module minor device* is required for each protocol being executed. Any number of protocol modules may be implemented, but no more than four can be in use at any one time since no more than four KMCs are supported. In general, each protocol module can have up to 256 minor devices. The VPM Release 2.0 protocol module, however, can have at most 16 minor devices; this restriction is due to the fact that the minor device number of the VPM protocol module is used not only to specify the VPM minor device but also to specify the interface-module minor device and the KMC minor device. The low-order four bits of the protocol-module minor device number determine the protocol-module minor device; the next two bits determine the interface-module minor device; the next two bits determine the KMC minor device.

Transmit buffers and receive buffers are passed between the VPM interpreter, the interface module, and the protocol module by means of pointers to data structures known as *buffer descriptors*. The buffer-descriptor structure is defined as follows:

```
struct vpmbd {
        short   c_ct;           /* Buffer size */
        short   d_adres;        /* Low-order 16 bits of buffer address */
        char    d_hbits;        /* High-order 2 bits of buffer address */
        char    d_type;         /* Protocol-dependent */
        char    d_sta;          /* Protocol-dependent */
        char    d_dev;          /* Protocol-dependent */
        struct buf *d_buf;      /* Pointer to system buffer descriptor */
        int     d_bos;          /* Index of next byte in buffer */
        int     d_vpmtdev;      /* Minor device number */
}
```

For empty receive buffers, *c_ct* must be equal to the buffer size in bytes; for transmit buffers, *c_ct* must be equal to the number of bytes to be transmitted. When a receive buffer is returned

---

* Strictly speaking, the interface module is not a driver and therefore does not have minor devices; however, the minor device number in this case selects an element of the data-structure array associated with the interface module in the same way that the minor device number associated with a driver selects an element of a data-structure array.

to the protocol module, *c_ct* is equal to the number of data bytes in the buffer. *D_adres* and *d_hbits* must contain an 18-bit UNIBUS-mapped buffer address; the low-order 16 bits must be in *d_adres* and the high-order two bits must be in the low-order two bits of *d_hbits*. *D_type*, *d_sta*, and *d_dev* are protocol-dependent; when using the BISYNC interpreter these three bytes may be read and modified by the protocol script. See the discussion of *getxbuf*, *getrbuf*, *rtnxbuf*, and *rtnrbuf* in the attached manual entry for *vpmc*(1C). *D_buf* contains a pointer to a system buffer descriptor; this is used to return the buffer to the system buffer pool. *D_bos* is the index of the first byte in the buffer not yet returned to the user. *D_vpmdev* is the minor device number of the protocol-module minor device to which the buffer is allocated.

The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver and the protocol script to aid in debugging new protocol modules and protocol scripts. It may also be used to debug other drivers or system code not related to the VPM driver. This driver can be configured to have a number of minor devices. Each minor device provides a means by which a user program can read data generated by functions within the operating system. This data is recorded by calls to *trsave* as described in Appendix 3. Each call to *trsave* generates a unit of data known as an *event record* which consists of a channel number (one byte), a count (one byte) and *count* bytes of data. The channel number can be used to multiplex up to 16 data streams on each minor device.

Associated with each minor device of the trace driver is a *clist* queue which is used to save event records provided a user program has that minor device open and has enabled the channel to which the event records were written. Channels may be enabled in any combination, using the *ioctl* command VPMTRCO. See the attached manual entry for *trace*(4). While a minor device read queue is full, event records for that minor device are discarded. Appendix 3 contains a description of each trace-driver routine.

Minor device 0 of the trace driver is used by the VPM driver to record a variety of debugging information generated within the VPM driver and also to record the data generated by the *trace* primitive in a protocol script. Minor device 1 of the trace driver is used to record the information generated by the *snap* primitive in a protocol script. The *vpmtrace* and *vpmsnap* commands are available for reading and formatting the data passed via these two minor devices. These two commands are described in the attached manual entry for *vpmstart*(1C). Appendix 4 contains a description of the VPM driver event trace.

Miscellaneous Improvements

Two new primitives have been added to the protocol language to allow communication between the link-level protocol script in the KMC and a higher-level protocol implemented in a user program or a VPM protocol module. The *getcmd* primitive allows the script to receive a four-byte command from a user program or a protocol module. The standard VPM protocol module allows a user program to pass a command to the script via an *ioctl* system call. Other VPM protocol modules can pass a command to the script by calling the *vpmcmd* routine in the VPM interface module. The *rtnrpt* primitive allows the script in the KMC to send a four-byte report to a protocol module or to a user program. The standard VPM protocol module allows a user program to receive a script report by means of an *ioctl* system call. A protocol module can receive reports from the interface module by calling the *vpmrpt* routine of the VPM interface module.

The *trace* primitive of the protocol language has been augmented to allow two arguments. The form with one argument is still supported; if only one argument is given, the second argument is assumed to be zero. A *snap* primitive has been added. This primitive causes four bytes of data from the script followed by a four-byte time stamp to be placed on the read queue for trace driver minor device 1.

The *timeout* primitive provided in Release 1.0 has been supplemented by a new *timer* primitive that allows a script to initialize a timer or test its current value. If the argument to *timer* is non-zero, the timer is initialized with the value of the argument. The timer is decremented ten times a second until the timer reaches zero. If the *timer* primitive is called with an argument of zero, it returns the current value of the timer. This value is zero if the timer has expired, otherwise non-zero.

In release 1.0 of VPM, the interpreter would accept at most one transmit buffer and one receive buffer at any given time. In Release 2.0 the interpreter will accept up to four transmit buffers and four receive buffers at a time. This applies to both the character-oriented (BISYNC) interpreter and the bit-oriented (HDLC) interpreter.

For applications with requirements for monitoring the integrity of the computer hardware and software, a form of cross-checking between the UNIX driver and the KMC has been implemented. Every three seconds the VPM interpreter in the KMC sends an "I'm-OK" report to the host; the host responds by sending an "I'm-OK" command to the KMC. If either the host or the KMC does not receive the "I'm-OK" signal within a reasonable time period, an error termination occurs.

Appendix 5 contains detailed instructions for adding VPM Release 2.0 to a UNIX 3.0 system. Appendix 6 describes a number of test programs and procedures that may be used to check the VPM hardware and software and to gain familiarity with the system.

Acknowledgements

We would like to thank our supervision, especially R. C. Haight and G. W. R. Luderer, for their support of the Virtual Protocol Machine. L. A. Wehr provided the initial idea of interpreting protocol descriptions with the KMC and helped us with debugging and useful advice from time to time. R. V. Baron of Department 9362 suggested a number of new features that became part of this release. R. M. Ermann of Department 5251 wrote the protocol script for the LAPB protocol and suggested several improvements in the HDLC primitives.

Finally, our thanks go to Mrs. R. J. Fiore for patiently typing and revising this document.

References

[1] Long, P. F. and Mee, C., III. "Release 1.0 of the UNIX Virtual Protocol Machine", TM 79-3624-6, Oct. 11, 1979.

[2] Ermann, R. M. "Formal Specification of X.25 Compatible Link Protocol", TM 79-5251-2, June 21, 1979.

[3] Ermann, R. M. "Portable Implementation of BX.25 Level 2", TM 80-5224-1, Sept. 17, 1980.

[4] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G. (eds.). *UNIX User's Manual*—Release 3.0, June 1980.

P. F. Long

MH-3673-PFL/CM3-rjf

C. Mee III

Atts.
Appendices 1-6
Manual entries

Appendix 1

The VPM Interface Module

The VPM interface functions provide a general-purpose interface between a higher-level protocol implemented in a VPM protocol module and the link-level protocol script executed by the VPM interpreter in the KMC. The KMC driver is used by the interface functions to pass commands to and receive reports from the VPM interpreter. When reports are received by the interface module that must be passed on to the protocol module, the protocol module's receive-interrupt routine (*vpmurint* in the case of the standard VPM protocol module) is called.

This appendix describes each interface function. *Dev* is an argument to many of the interface functions and has the same meaning for all but two of them: the low-order four bits of the argument are *not* used by the interface functions; the next two bits determine the interface module minor device number; the next two bits determine the KMC minor device. Although *dev* is declared as an *int*, only the low-order eight bits are meaningful at this time. In calls to the *vpmtrace* and *vpmsnap* routines, *dev* need not be a minor device number since it is just saved as part of the event record. The definition of *dev* will not be repeated for each function.

```
vpmcmd (dev, cmd)
int dev;
char *cmd;
```

This function passes a command to the script. *Cmd* is the address of a four-byte array. The four bytes are passed to the VPM interpreter, which saves them until the protocol script executes a *getcmd* primitive. Only the most recent four bytes passed by a *vpmcmd* call are saved by the VPM interpreter.

```
struct vpmbd *vpmdeq (clp)
struct clist *clp;
```

This function removes the buffer-descriptor pointer at the head of the queue pointed to by *clp* and returns it to the caller. If the queue is empty, a null pointer is returned.

```
vpmemptq (dev, bdp)
int dev;
struct vpmbd *bdp;
```

This function is used to pass an empty receive buffer for use by the interpreter in the KMC. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer descriptor is appended to the empty-receive-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another empty receive buffer, the buffer at the head of the queue is removed and passed to the KMC. The sum of the number of buffers on the empty-receive buffer queue and the number of receive buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

```
vpmenq (bdp, clp)
struct vpmbd *bdp;
struct clist *clp;
```

If *bdp* is a null pointer, the number of buffer-descriptor pointers on the *clist* queue pointed to by *clp* is returned. If *bdp* is not a null pointer, the buffer descriptor pointed to by *bdp* is appended to the *clist* queue pointed to by *clp* and the number of pointers currently on that queue is passed as the return value.

```
char *vpmerrs (dev, n)
int dev, n;
```

This function is used to read and reset the error counters in the VPM interpreter. The function

passes a GETECMD command to the VPM interpreter and blocks until the interpreter responds; this command causes the interpreter to copy its error counters to an array in the interface module and send a completion report to the driver. After the copy operation is completed, a pointer to the error-count array is passed to the caller as the return value. The second argument is not currently used.

```
char *vpmrpt (dev)
int dev;
```

This function is used to receive a script report from the KMC. When the protocol script executes a *rtnrpt* primitive, four bytes of data are passed to the interface module. If a *rtnrpt* has been executed by the protocol script since the last call to *vpmrpt*, a pointer to the four bytes passed by the most recent *rtnrpt* primitive is returned; otherwise zero is returned.

```
vpmsave (type, dev, word1, word2)
char type, dev;
short word1, word2;
```

This function creates an event record with the following structure:

```
struct {
        short   c_seqn;         /* Sequence number */
        char    c_type;         /* Argument type */
        char    c_dev;          /* Argument dev */
        short   c_word1;        /* Argument word1 */
        short   c_word2;        /* Argument word2 */
}
```

This event record is passed to the *trace* driver using *trsave*.

```
vpmsnap (type, dev, word1, word2)
char type, dev;
short word1, word2;
```

This function is similar to *vpmsave*. The only difference is that a time stamp *(long s_lbolt)* is added to the event record after *word2*. A protocol script may generate a time-stamped event record by executing the *snap* primitive.

```
vpmstart (dev, type, rint)
int dev, type;
int (*rint)();
```

This function must be called on the first open of the protocol-module minor device associated with the interface-module minor device and KMC identified by *dev*. *Type* is a number that identifies the program running in the KMC and must agree with the value specified when the KMC load module was loaded into the KMC. For VPM interpreters, *type* is conventionally 6. *Rint* is the name of a protocol-module routine to be called by the interface module when it needs to return a transmit buffer, a receive buffer, a script report, or an error-termination code. See the description of *vpmrint* in appendix 2 for an example of such a routine. *Vpmstart* sends a RUN command to the VPM interpreter which causes it to begin execution of the protocol script. If the interface module identified by *dev* is not configured, ENXIO is returned. If the module is already running, i.e., *vpmstart* has been called and *vpmstop* has not been called, or if the KMC is not running or was loaded using a different magic number, EACCESS is returned. A return value of zero indicates a normal completion.

```
vpmstop (dev)
int dev;
```

This routine is called to halt the execution of the protocol script by the interpreter. The routine waits until the last transmit buffer has been returned by the protocol *script*, or until five seconds

have elapsed, and then sends a HALT command to the VPM interpreter which causes the interpreter to stop executing the protocol script. When the interpreter acknowledges the HALT command, or after five seconds, any transmit or receive buffers still enqueued on the interface module's transmit- and empty-buffer queues are returned to the protocol module. This does not include buffers contained in the interpreter's queues. Generally, when the protocol script is halted normally, the interpreter will have one or more empty receive buffers. If the interpreter or protocol script terminates in error, some transmit buffers may also remain unaccounted for. The upshot of this is that a protocol module must keep a record of all buffers in use for each particular minor device, so that these buffers can be returned to the pool of available buffers when that minor device is closed.

Appendix 2

The VPM Protocol Module

This appendix gives a detailed description of the functions that make up the standard VPM protocol module. The description may be useful as a guide in writing other VPM protocol modules. The *dev* argument to the following routines is declared as an *int;* however, only the low-order eight bits are meaningful at this time. The low-order four bits are used to determine the minor device of the protocol module; the next two bits determine the minor device of the interface module; the next two bits determine the KMC minor device.

```
vpmopen (dev, flag)
int dev, flag;
```

This function opens the protocol-module minor device specified by the low-order four bits of *dev*. *Flag* contains the option bits specified on the *open* system call. Exclusive or non-exclusive *open*s are permitted. If the driver is opened for both reading-and-writing, the *open* is exclusive, i.e., no further *opens* are permitted. If the device is opened for reading only or for writing only, the *open* is non-exclusive and subsequent *open*s for reading only or writing only are permitted. If this device is not open when this function is called, it obtains a number of non-addressable system buffers to be used as receive buffers and passes them to the VPM interpreter using the interface routine *vpmemptq*. *Vpmopen* also calls the interface routine *vpmstart* if the minor device was not already open.

```
vpmclose (dev)
int dev;
```

This function closes the minor device specified by the low-order four bits of *dev*. It calls the interface routine *vpmstop*, flushes the receive queue for the specified minor device, releases its buffers, and reinitializes its data structure.

```
vpmwrite (dev)
int dev;
```

This function implements the *write* system call. If the transmit queue is not full, the function obtains a non-addressable system buffer, copies up to 512 bytes of the user's *write* data into it, and enqueues the buffer on the level 2 transmit queue using the interface function *vpmxmtq*. These steps are repeated until all of the user's *write* data has been copied. If the transmit queue is full when this function is called or if it becomes full while the function is executing, the calling process is blocked until there is room in the queue for more transmit buffers.

```
vpmread (dev)
int dev;
```

This function implements the *read* system call. When it is called, the calling process is blocked until the receive queue is non-empty. As data is received by the VPM interpreter, it is placed into an empty receive buffer. When the protocol script decides that the data contained in a particular buffer is valid, it executes a *rtnrbuf* (BISYNC) or *rtnrfrm* (HDLC) primitive which causes the buffer descriptor pointer to be passed to the interface modules interrupt routine. The interface module then passes the buffer descriptor pointer to the protocol module by calling the protocol module's interrupt routine. The protocol module enqueues the buffer descriptor pointer on the receive queue and wakes up (unblocks) the reader(s). The number of bytes requested, or the data in one buffer, whichever is less, is copied to the user process; the number of bytes copied is passed as the return value. Any bytes remaining in a buffer are used to satisfy subsequent *read* requests.

vpmioctl (dev, cmd, arg, mode)
int dev, cmd, mode;
char *arg;

This function implements the *ioctl* system call. *Cmd* determines the function to be performed as follows:

VPMCMD - Pass a command to the protocol script. The first four bytes of the array pointed to by *arg* are passed to the VPM interpreter which saves them and passes them to the protocol script the next time it executes a *getcmd* primitive.

VPMERRS - Get and reset the VPM interpreter's error counters. The eight-byte array containing the VPM interpreter's error counters is copied to the user array pointed to by *arg*. The interpreter's copy of the error counters is then set to zero.

VPMRPT - Get a report from the protocol script. If the protocol script has executed a *rtnrpt* primitive since the last time this *ioctl* command was issued, the script report (four bytes) is copied to the user array pointed to by *arg* and *one* is passed as the return value; otherwise, *zero* is passed as the returned value.

The *node* argument is not used. The values for VPMCMD, VPMERRS, and VPMRPT are defined in file */usr/include/sys/vpm.h*.

vpmtrint (dev, code, bdp)
int dev, code;
struct vpmbd *bdp;

The address of this function is passed to the protocol module using the *vpmstart* function described in Appendix 1. This routine is called from the interface module to return transmit buffers, receive buffers, script reports, or error termination codes. It is usually called at interrupt priority and therefore must not sleep or do unnecessary work. *Code* identifies the purpose of the call and determines the meaning of *bdp* as follows:

RRTNXBUF - *Bdp* is a pointer to the buffer descriptor for a transmit buffer. This call is made when the protocol script executes a *rtnxbuf* (BISYNC) or a *rtnxfrm* (HDLC).

RRTNRBUF - *Bdp* is a pointer to the buffer descriptor for a receive buffer. This call is made when the protocol script executes a *rtnrbuf* (BISYNC) or a *rtnrfrm* (HDLC).

RRTNEBUF - *Bdp* is a pointer to the buffer descriptor for an empty receive buffer. This call is used to return empty receive buffers when the interface module is stopped by calling *vpmstop*.

ERRTERM - *Bdp* is the error-termination code passed to the interface module by the VPM interpreter when it halts the protocol script because of an error condition. The meaning of these error codes is given in the attached manual entry for *vpm*(4).

The values for RRTNXBUF, RRTNRBUF, RRTNEBUF, and ERRTERM are defined in file */usr/include/sys/vpm.h*.

Appendix 3

The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver, a protocol script, or some other driver. See the attached manual entry for *trace*(4).

A description of each routine of the trace driver follows.

tropen (dev)
int dev;

This function opens the minor device specified by *dev* exclusively.

trclose (dev)
int dev;

This function closes the minor device specified by *dev*. It discards any data on the read queue and initializes the data structure associated with the minor device.

trread (dev)
int dev;

This function implements the *read* system call; it sleeps until at least one event record is available on the read queue associated with *dev*. It then copies records to the user until the user's read count is less than the number of bytes in the next event record or until the read queue is empty. The number of bytes copied is passed as the return value.

trioctl (dev, cmd, arg, mode)
int dev, cmd, arg, mode;

This function implements the *ioctl* system call. *Cmd* indicates the operation to be performed. The driver has one command:

> VPMTRCO - Enable a trace channel. In order for data to be saved on the read queue for minor device *dev*, the device must be open and the channel to which it is written must be enabled. This command enables channel *arg*, which must be in the range 0 to 15. Any combination of channels may be enabled by repeatedly calling this function with different values of *arg*. All channels are disabled when the minor device is closed.

trsave (dev, chno, buf, ct)
char dev, chno, *buf, ct;

If minor device *dev* of the trace driver is open and channel *chno* of that minor device is currently enabled then *chno* and *ct*, followed by *ct* bytes starting at address *buf*, are copied onto the read queue associated with *dev*, provided the read queue for that device has room for the complete event record. If there is not room for the complete event record, the record is discarded.

Appendix 4

The VPM Event Trace

Calls to the interface routine *vpmsave* have been placed strategically throughout the standard VPM protocol module (*vpmt.c*) and the VPM interface module (*vpmb.c*) to provide an event trace for debugging new protocol modules and/or protocol scripts. A protocol script may generate an event record by executing a *trace* primitive. All such event records are discarded unless some user program has opened minor device 0 of the trace driver and enabled channel 0 of that minor device. The command *vpmtrace*(1C) opens this device and enables channel 0, then reads event records and prints them on the standard output as they are received. Each kind of event record that is generated by the VPM driver will be described by giving the *vpmsave* function call as it appears in *vpmt.c* or *vpmb.c*, followed by an example of the line printed by *vpmtrace* as a result of this call. Following this, the context of the *vpmsave* call and the definition of the parameters passed will be given. The definition of a parameter that appears in more than one call will not be repeated. The first five calls to *vpmsave* occur in the source file *vpmt.c*; the remaining calls occur in *vpmb.c*.

vpmsave ('p', dev, ec, 0)

243 p 100 15 0

Called if *vpmstart* returns an error code. The first field of the printed record contain a sequence number assigned by *vpmsave*. The remaining four fields contain the four remaining arguments to *vpmsave* in the same order as they appear in the call to *vpmsave*. The first argument to *vpmsave*, in this case a 'p', identifies the record type. *Dev* is the minor device number as defined earlier; *ec* is the value returned by *vpmstart*.

vpmsave ('o', dev, vp->vt_state, 0)

244 o 100 1 0

Called just before the normal return point of *vpmopen*. The variable, *vp->vt_state*, contains the state bits for the protocol module. Refer to the source file, *vpmt.c*, for the definitions of the state bits.

vpmsave ('c', dev, vp->vt_state, 0)

245 c 100 13 0

Called from *vpmclose* just before the state bits are initialized.

vpmsave ('w', dev, ct, dp)

246 w 100 1000

Called just before putting a buffer-descriptor pointer on the transmit queue in *vpmwrite*. *Ct* is the number of bytes in the buffer. When executing on a PDP11, *dp* is the pointer to the buffer descriptor; *dp* is not meaningful when executing on a VAX because pointers are four bytes on a VAX and the argument corresponding to *dp* is declared as a *short*.

vpmsave ('r', dev, cnt, dp->d_bos)

247 r 100 500 500

Called from *vpmread* just after *cnt* bytes have been moved to the user's read buffer. The parameter *dp->d_bos* is the number of bytes remaining in the current receive buffer.

vpmsave ('s', dev, vp->vb_state, 0)

248 s 100 401 0

Called just before the normal return from *vpmstart*. The parameter *vp->vb_state* contains the state bits for the interface module. For the definitions of the state bits, refer to the source file

vpmb.c.

vpmsave ('t', dev, vp->vb_state, vp->vb_xbkmc)

249 t 100 0 0

Called just before the normal return from *vpmstop*. The parameter *vp->vb_xbkmc* is the number of transmit buffers currently held by the VPM interpreter. It can be non-zero if the protocol script or interpreter terminates in error.

vpmsave ('X', dev, vp->vb_xbkmc, 0)

250 X 100 1 0

Called from *vpmbrint*, the interface module's receive-interrupt routine, each time the VPM interpreter returns a transmit buffer.

vpmsave ('R', dev, vp->vb_vrkmc, 0)

251 R 100 1 0

Called from *vpmbrint* each time the VPM interpreter returns a receive buffer. The parameter *vp->vb_rbkmc* contains the number of receive buffers currently held by the interpreter.

vpmsave ('T', dev, sel4, sel6)

252 T 100 370 21 34

Called from *vpmbrint* when a *trace* report is received from the interpreter. This occurs when the protocol script executes a *trace* primitive. *Sel4* contains the value of the script location counter (plus two) at the time the *trace* primitive was executed. By referring to the assembly-language listing of the protocol script generated by the *-l* option of *vpmc*, the point in the protocol script at which the *trace* was executed can be determined. The value of the location counter is two greater than the location of the *trace* instruction as shown in the assembly-language listing. *Sel6* contains the byte or bytes passed by the *trace* primitive. *Vpmtrace* prints these two bytes in separate fields.

vpmsave ('E', dev, sel4, sel6)

253 E 244 21

Called from *vpmbrint* when an error-termination report is received from the interpreter. *Sel4* contains the script location counter at the time execution of the script was terminated. *Sel6* contains the termination code. For an explanation of these codes see the attached manual entry for *vpm*(4).

vpmsave ('P', dev, sel4, sel6)

254 P 100 2105 1055

Called from *vpmbrint* when a script report is received from the interpreter. This occurs when the protocol script executes a *rtnrpt* primitive. *Sel4* and *sel6* contain the four bytes transferred by this primitive.

vpmsave ('F', dev, sel4, sel6)

255 F 100 3 0

Called from *vpmbrint* when an error-count report is received from the interpreter. *Sel4* and *sel6* do not contain any meaningful data for this event type.

vpmsave ('S', dev, sel4, sel6)

256 S 100 401 0

Called from *vpmbrint* when a start-up report is received from the interpreter. The low-order

eight bits of *sel4* contain a parameter defining the maximum number of transmit buffers the interpreter can accept; the high-order eight bits contain a parameter defining the maximum number of receive buffers. *Sel6* contains the options supported by the interpreter.

vpmsave ('C', dev, vp->vb_state, bp->vb_xbkmc)

257 C 100 1 0

Called from *vpmclean* just before the data structure associated with *dev* is initialized.

vpmsave ('O', dev, vp->vb_state, 0)

258 0 100 1 0

Called from *vpmok* if the interpreter should fail to indicate its sanity by issuing an "I'm-OK" report within the prescribed time limit.

Appendix 5

Adding VPM to a UNIX Release 3.0 System

The UNIX Release 3.0 distribution tapes contain VPM Release 2.0. This includes the compiler, drivers, interpreters, utility commands, protocol scripts, and test programs.

The *makefile vpm.mk* found in */usr/src/cmd/vpm* may be used to make and install all VPM commands.

To add the VPM and trace drivers to a UNIX 3.0 system, do the following:

1. Make sure that the following two lines appear in the file */etc/master:*

```
vpm      0   37   206   vpm   0   0     15    16    5
trace    0   35   206   tr    0   0     16    4     1
```

2. Add the following line to the file */usr/src/uts/*/cf/cfigpa* (or its equivalent):

```
vpm      0   0    0     n
```

   where *n* is the number of minor devices required. The * represents either *pdp11* or *vax*.

3. To the same file add the following line for each trace minor device:

```
trace    0   0    0     n
```

   where *n* is the number of minor devices required. Minor device 0 is used by the *vpmtrace* command and minor device 1 is used by *vpmsnap*.

4. If KMCs are being added to the system, add the following line to the same file for each KMC:

```
kmc11    vector    address    priority
```

   where *vector* is the interrupt vector location (octal), *address* is the device address (octal), and *priority* is the bus request level (normally 5).

A special file must be created in */dev* for each KMC, VPM, and trace device. To make these special files, use *mknod*(1M) as follows:

For KMCs:

/etc/mknod /dev/kmc? c X ?

where X is the major device number of the KMC driver as printed by *config* -t (see the manual entry for *config*(1M)[4]) and ? is the minor device number which must be in the range 0 to 3.

For VPMs:

/etc/mknod /dev/vpm c Y Z

where *vpm* is a unique device name; Y is the major device number of the VPM driver; and Z is a decimal or octal number whose binary representation is defined as follows: the low-order four bits specify one of up to 16 minor devices of the standard VPM protocol module; the next two bits specify one of up to four VPM interface-module minor devices; the next two bits specify the minor device number of the KMC to be used for this special file.

For trace devices:

```
/etc/mknod    /dev/trace   c Y 0
/etc/mknod    /dev/snap    c Y 1
```

where Y is the major device number of the trace driver.

Hardware Installation and Switch Settings

The KMC11-B microprocessor and DMC11-DA, -FA, or -FD line unit must be installed in adjacent slots of a PDP-11 or VAX-11/780 backplane. Care should be taken not to exceed the DC power capacity of the cabinet in which the items are installed. The microprocessor and line unit are interconnected by a one-foot ribbon cable. The line unit is connected to a suitable synchronous modem by a 25-foot modem cable. If the HDLC interpreter is used, the modem must be optioned for full-duplex (four-wire) operation; at speeds above 1200 bits per second this will normally require a private line. The device address and interrupt vector address switches on the KMC should be set for the selected addresses. The KMC should also be wired for the selected bus priority (normally 5). All switches and jumpers on the DMC line unit should be in the normal configuration prescribed by the relevant DEC maintenance manual with one exception: the NO CRC switch (switch S2 in switch pack number 1) should be in the ON position. The purpose of this switch setting is to inhibit hardware CRC generation. Hardware CRC generation is not used with the VPM software for this device.

If the KMC is a Revision E, a DEC field change is required before it can be used with VPM.

Appendix 6

Testing VPM

During the course of developing and testing VPM, a number of programs and test procedures have evolved which may prove useful to those adding VPM to a system or using VPM for the first time. These programs and procedures will help to check the correct installation and operation of the hardware and software as well as help a new user of VPM to gain familiarity with the package. These programs may be found in */usr/src/cmd/vpm/demo* and */usr/src/cmd/vpm/scripts*.

Decbin

Decbin is a simple KMC program that exercises enough of the KMC memory and instruction set so that a correct result provides reasonable assurance that the KMC is functioning properly. It does *not* exercise the interface between the KMC and the DMC11 line unit.

To run this test, you must compile file *decbin.k* in directory */usr/src/cmd/vpm/demo*. This can be done as follows:

```
/lib/cpp  /usr/src/cmd/vpm/demo/decbin.k | kasb
```

You must then load and run the resulting *a.out* and then dump the KMC and its registers. The following sequence of commands will accomplish this:

```
kasb -d /dev/kmc?
.reset
.load
.run
.reset
.dump
.regs
```

The *.regs* command to *kasb* will produce a register dump similar to the following:

```
csr:     377   0   0   0   0   0   0  20
lur:       0  20   0 101   0 377 377  53
reg:       0 326  42  64   0 276   0  46
reg:     142  73 321  71 156  61 116 356
 io:     377 377 377 377 377 371 377 377
npr:       0  20   0 brg: 356   0 mem:  61
```

If the value of r5 (the sixth number in line three of the register dump) is not 276, something is wrong with the KMC hardware or the software used to load and execute programs in it.

Tset

Tset is a C program that opens a particular *vpm* device (*/dev/vpm0*) and writes a string of characters to it. It then reads the same device and compares the string of characters received to the string sent. If the two strings match, the program prints the string followed by the message "It worked!!!!!!". This program will work only when a loopback script such as *loop.r* has been loaded into the KMC. To run this test:

1. Compile *tset.c:*

   ```
   cc -o tset tset.c
   ```

2. Compile *loop.r:*

   ```
   vpmc -o loop.o loop.r
   ```

3. Load *loop.o* into the KMC:

```
/etc/vpmstart /dev/kmc? 6 loop.o
```

4. If testing the VPM event-tracing capability, execute *vpmtrace:*

```
/etc/vpmtrace > t&
```

5. Execute *tset:*

```
tset
```

6. Print *t:*

```
cat t
```

Sr

Sr opens */dev/vpm0* and forks to create a *send* process and a *receive* process. The *send* process reads up to 512 bytes from its standard input and writes them to */dev/vpm0*. The receive process reads */dev/vpm0* and writes the received data to its standard output. This program may be used with the protocol script *loop.r*. The procedure for running *sr* is similar to that used with *tset*. Steps 2, 3, and 4 need not be repeated if the interpreter and *vpmtrace* are still running.

To execute *sr:*

```
sr < infile > outfile
```

The *send* process exits after it has read and transmitted the last data block of the file. The *receive* process goes into a loop that sets an alarm and reads */dev/vpm0*. If the alarm goes off before the *read* completes, the process exits.

Tcmd

Tcmd.c when used with the protocol script *tcmd.r* tests several new features of Release 2.0 of VPM: communications between a user program or a protocol module and the protocol script, reading and resetting the interpreter's error counters, and the time-stamped tracing capability. To execute *tcmd*, follow the procedures given for the first test using *tcmd.c* and *tcmd.r* in place of *tset.c* and *loop.r*. Execute *vpmsnap* instead of or in addition to *vpmtrace*.

Lapb.r

Lapb.r is the protocol script for BX.25 Level 2. To install this script in a particular KMC, proceed as follows:

```
cp /usr/src/cmd/vpm/scripts/lapb.r .
cp /usr/src/cmd/vpm/scripts/const .
cp /usr/src/cmd/vpm/scripts/tconst .
vpmc -mi hdlc -o lapb.o lapb.r
/etc/vpmstart /dev/kmc? 6 lapb.o
```

Testing this script requires two KMCs, which may be on different host computers. The KMCs must be connected by a pair of full-duplex synchronous modems or by a full-duplex synchronous null modem.* *Sr* should be executed simultaneously on both machines to read and write the VPM device associated with each KMC. If both KMCs are on the same host machine, it will be necessary to edit and compile a copy of *sr.c* so that it opens */dev/vpm1* instead of */dev/vpm0*. The original and modified versions of *sr* can then be executed simultaneously to exercise the two KMCs.

---

* A suitable null modem is the Avanti 300, which is manufactured by Avanti Communications Corporation, Newport, RI.

To obtain maximum efficiency from this script, it may be necessary to modify the values of some of the parameters in the *const* file. The appropriate values for these parameters depend on the link speed and maximum frame size. Guidelines for adjusting these parameters are given in [3].

Lapbt.r

This script is identical to *lapb.r* except for some additional *trace* statements. It may be tested in the same manner as *lapb.r*. *Vpmtrace* may be used to display the trace information.

Itr.r

Itr.r is a simplified version of *lapb.r*. Unlike *lapb.r* and *lapbt.r*, this script can be exercised in a loop-back mode. To run a loop-back test, attach a DEC H-325 test connector to the end of the modem cable for the DMC11-DA line unit that is connected to the KMC11-B to be used for the test. Then compile *itr.r* and load the resulting *a.out* into the KMC using the procedure described above for *lapb.r*, substituting *itr* for *lapb*. A loop-back test can then be run using *tset* or *sr*.

Setting Up UNIX

R. C. Haight
T. J. Kowalski
M. J. Petrella
L. A. Wehr

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Prerequisites

Before attempting to generate a UNIX† system, you should understand that a considerable knowledge of the related documentation is required and assumed. In particular, you should be very familiar with the following documents:

- *The UNIX Time-Sharing System*
- *UNIX User's Manual*
- *C Reference Manual*
- *Administrative Advice for UNIX*
- *UNIX Operations Manual*

A complete set of pertinent documentation is contained in *Documents for UNIX*. Throughout this document, each reference of the form *name*(N), where N is a number, refers to entry *name* in Section N of the *UNIX User's Manual*.

You must have a basic understanding of the operation of the hardware. This includes the console panel, the tape drives, and the disk drives, all of which are assumed to have standard UNIX addresses and interrupt vectors. It is also assumed that the hardware works and has been completely installed. All appropriate DEC® diagnostics should have been run to test the configuration, and you must have a detailed description of the hardware, including device addresses, interrupt vectors, and bus levels. This information is necessary to generate a UNIX system.

See Attachment 1 for a list of supported CPUs and devices.

1.2 Procedure

UNIX is distributed on two magnetic tapes, recorded in 9-track format at 800 bpi. Distribution tapes will be marked either "PDP-11" or "VAX"; be sure you have the correct tape for your machine.

The initial load program (on Tape 1) will copy a file system from tape (VAX: TE16; PDP-11: either a TU10 or a TU16) to disk (VAX: RP06; PDP-11: either an RP03, an RK05, an RL01, or an RP06). In this document, we consider RP04 and RP05 drives to be equivalent to RP06 drives; any differences will be noted explicitly. Once the *root* file system has been successfully loaded to disk, UNIX may be booted and the available utility programs may then be used to complete the installation.

In order for any of the update procedures to work correctly, you *must* be *running* UNIX/TS or PWB/UNIX Release 2.0. Older versions of UNIX cannot be correctly *updated* with a UNIX system. The *cpio*(1) program will not replace any file if its replacement has a modification time that is less than (i.e., earlier than) the modification time of the original file. This can be due to local modifications. Furthermore, certain administrative files (e.g., */etc/passwd*, */usr/lib/crontab*)

---

† UNIX is a Trademark of Bell Laboratories.

are sent with a modification time of January 1, 1970 to ensure that they do not replace their counterparts during updates. Any file not copied will cause *cpio* to print a message to that effect. These messages should always be investigated to ensure that any files not copied were of that type. However, note that, depending on respective modification times, a locally-modified file may get updated, thus destroying the local modifications.

There are several difficulties that can arise when installing a UNIX system. One of the most common problems is running out of disk space when performing an update. Should this occur, the original contents of the file system should be restored from a backup copy and the contents of the update tape should be read into a spare file system using the *cpio* program. Unwanted material can then be removed and the original file system can be updated from this new file system using the **-p** option of *cpio*. Modification times of files should also be preserved using the **-m** option of *cpio*.

2. LOAD PROCEDURES

2.1 Distribution Tape Format

Tape 1 contains seven files: a loader, a physical copy of the *root* file system, the *cpio* program, a *cpio* structured copy of the *root* file system, and three files (*cpio* format) that represent selectable items. *Root* refers to the directory "/", which is the root of all the directory trees. The format of this tape is as follows:

record 0:	Tape boot loader– 512-bytes;
record 1:	Tape boot loader– 512-bytes;
remainder of file 1:	Initial load program– several 512-byte records;
end-of-file	
file 2:	*root* file system (physical)– 5,120-byte records (blocking factor 10);
end-of-file	
file 3:	*cpio* program (latest version)– several 512-byte records;
end-of-file	
file 4:	*root* file system (structured in *cpio* format)– several 5,120-byte records (to be used *only* for updating an earlier UNIX);
end-of-file	
file 5:	on-line documentation (same format as file 4);
end-of-file	
file 6:	rje software (same format as file 4);
end-of-file	
file 7:	graphics software (same format as file 4).
end-of-file	

The *root* (/) file system contains the following directories:

bck:	Directory used to mount a backup file system for file restoring.
bin:	Public commands; most of what is described in Section 1 of the *UNIX User's Manual*.
dev:	Special files, all the devices on the system.
etc:	Administrative programs and tables.
lib:	Public libraries, parts of the assembler, C compiler.
mnt:	Directory used to mount a file system.
lost+found:	Directory used by *fsck*(1M) for disconnected files.
stand:	Stand-alone programs.
tmp:	Directory used for temporary files; should be cleaned at reboot.
usr:	Directory used to mount the */usr* file system.

2.2 Initial Load of root

Mount Tape 1 on drive 0 and position it at the load point.

2.2.1 PDP-11

Bootstrap the tape by reading either record 0 or record 1 into memory starting at address 0 and start execution at address 0. This may be accomplished by using a standard DEC ROM bootstrap loader, a special ROM, or some manual procedure; see *romboot*(8), *tapeboot*(8), and *70boot*(8).

2.2.2 VAX

See "Installation Boot Procedures" under *vaxops*(8). This *UNIX User's Manual* entry describes initial tape booting, and modification of the console floppy disk to simplify UNIX administration.

2.3 Common to PDP-11 and VAX

The tape boot loader will then type "UNIX tape boot loader" on the console terminal and read in and execute the initial load program. The program will then type detailed instructions about the operation of the program on the console terminal. The program will ask what type of disk drive you have and which drive you plan to use for the copy. The disk controller used must be at the standard DEC address indicated by the program. However, other disk controllers on your system may be at non-standard addresses. You must mount a formatted, ECC flag-free pack on the drive you have indicated. If necessary, use the appropriate DEC diagnostic program to format the pack. Note that the pack will be written on. Next, the program will ask what type of tape drive you have and which drive contains the tape. Normally, this will be drive 0, but the program will work with other drives. Note that the tape is currently positioned correctly after the end-of-file between the initial load program and the *root* file system. When everything is ready, the program will copy the file system from the tape to disk and give instructions for booting UNIX. After the copy is complete and you have booted the basic version of UNIX, check (using *fsck*(1M)) the *root* file system and browse through it.

2.3.1 PDP-11/70 Only

The file */stand/mmtest* is a stand-alone memory mapping diagnostic program for the PDP-11/70. If you are not *absolutely* sure that DEC FCO (field change order) M8140-R002 has been applied to your PDP-11/70 CPU, *stand/mmtest* should be booted and allowed to run at least 20 minutes. To boot this program, go through the boot procedure, but specify:

```
0= stand/mmtest
```

UNIX Release 3.0 comes with optional power-fail recovery. This feature *requires* that the *power-up* and *power-down* interrupt vectors be 024.

2.4 Update of root

It is very important that the system be running in *single-user* mode during the update phase. To update an already existing *root* file system, files three and four on Tape 1 will be used. It is necessary to first make a copy of your *root* file system using *volcopy*(1M) and then update this copy. The copy should be made on a separate disk pack using the same section number as your *root* file system (always section 0). Also, after the update is completed, check if any of your local administrative files in the directory */etc* need modification. Most of these are mentioned in Section 4 below.

Mount Tape 1 on drive 0 and position it at the load point. We assume that disk drive 1 is available for making the copy, and that the *root* file system is on */dev/rp0*. This shell procedure will be different for RK05 and RL01 disk drives. The following procedure will first make a copy of the *root* file system, and then update this copy. Note that */dev/mt4* refers to tape drive 0 but has the side effect of spacing forward to the next end-of-file (no rewind option). The −B

option of *cpio* specifies that input is in 5,120-byte records:

```
volcopy root /dev/rrp0 pkname1 /dev/rrp10 pkname2
mount /dev/rp10 /bck
#   The 2 echoes will move the tape to file 3
echo </dev/mt4
echo </dev/mt4
cp /dev/mt4 /bck/bin/cpio
chmod 755 /bck/bin/cpio
chown bin /bck/bin/cpio
cd /bck
/bck/bin/cpio -idmB </dev/rmt0
cd /
umount /dev/rp10
```

Pkname1 and *pkname2* are the volume names of the source and destination disk packs, respectively. If the new copy is satisfactory, shut down and halt the system, change disk packs, and reboot the system using the new *root*.

2.5 Tape 2 (/usr) Format

Tape 2 contains the */usr* file system in *cpio* format (5,120-byte records). The */usr* file system contains commands and files that must be available (mounted) when the system is in *multi-user* mode. The tape contains the following directories:

adm:	Miscellaneous administrative command and data files, including the connect-time accounting file *wtmp* and the process accounting file *pacct*.
bin:	Public commands; an overflow for */bin*.
dict:	Dictionaries for word processing programs.
games:	Various demonstration and instructional programs.
include:	Public C language *#include* files.
lib:	Archive libraries, including the text processing macros; also contains data files for various programs, such as *spell*(1) and *cron*(1M).
mail:	Mail directory.
man:	Source for the *UNIX User's Manual*; see *man*(1).
lost+found:	Directory used by *fsck*(1M) for disconnected files.
news:	Place for all the various system news.
pub:	Handy public information, e.g., table of ASCII characters.
spool:	Spool directory for daemons.
src:	Source for commands, libraries, the system, etc.
tmp:	Directory for temporary files; should be cleaned at reboot.

A table of contents of this tape may be obtained by using the *cpio* options −itB. Also, after installation, files and directories deemed useless by the local administrator may be easily removed. Alternately, only parts of the tape may be extracted using the pattern matching capabilities of *cpio*.

2.6 Initial Load of /usr

Mount Tape 2 on drive 0 and position it at the load point. Mount a file system (device) as */usr*. The ultimate size and location of this file system on a device is an administrative decision; initially, the following procedure will suffice:

```
mkfs /dev/rrp1 50000 7 418
labelit /dev/rrp1 usr pkname
mount /dev/rp1 /usr
chmod 775 /usr
cd /usr
cpio -idmB </dev/rmt0
```

Pkname is the volume name of the pack (e.g., "p0001").

Because */usr* must be mounted when the system is in *multi-user* mode, the file */etc/rc* must be changed to include the command lines to mount and unmount the file systems in *single-user* and *multi-user* mode. These lines must be inserted at the appropriate places in */etc/rc*, as indicated by comments in the prototype file. Next the file */etc/checklist* should be changed to include */dev/rrp1*, */dev/rrl1*, or */dev/rrk1*; see also *fsck*(1M), *labelit*(1M), *mkfs*(1M), *mount*(1M), and *checklist*(5).

2.7 Update of /usr

It is advisable that the system be running in *single-user* mode during the update phase. It is also wise to first make a copy of your */usr* file system for backup purposes. Next, mount Tape 2 on drive 0 and position it at the load point. The */usr* file system *must* also be mounted. The following procedure will perform the update:

```
cd /usr
cpio -idmB </dev/rmt0
```

2.8 Initial Load or Update of Selectable Items

The initial load and update procedures are essentially the same; the only exception being the creation of the selectable item directory on the initial load.

Mount Tape 1 on drive 0 and position it at the load point. Make sure that the */usr* file system is mounted. The following procedure will read in the source for each of the selectable subsystems. If a particular subsystem is not desired, simply skip that file on the tape by executing the following command:

```
echo </dev/mt4
```

The tape can be rewound after any subsystem by specifying */dev/rmt0* instead of */dev/rmt4*.

```
echo </dev/mt4
echo </dev/mt4
echo </dev/mt4
echo </dev/mt4
cd /usr
mkdir man; chown bin man; chgrp bin man; chmod 775 man
cd man
cpio -idmB </dev/rmt4

cd /usr/src/cmd
mkdir rje; chown bin rje; chgrp bin rje; chmod 775 rje
cd rje
cpio -idmB < /dev/rmt4

cd /usr/src/cmd
mkdir graf; chown bin graf; chgrp bin graf; chmod 775 graf
cd graf
cpio -idmB </dev/rmt0
```

After installing the source for the *rje* and *graphics* subsystems, the software must be *built* and *installed*. (Only execute the following commands for the subsystems that you have elected to take.)

To build and install graphics:

```
cd /usr/src
./:mkcmd graf
```

To build and install *rje*, select the appropriate *./:mkcmd* lines:

```
cd /usr/src
ARGS="dqsrje hasp"  ./:mkcmd rje  (makes a single DQS11-based hasp system)
ARGS="dqsrje hasp2" ./:mkcmd rje  (makes hasp and hasp2)
ARGS="dqsrje hasp3" ./:mkcmd rje  (makes hasp, hasp2, and hasp3)
ARGS="dqsrje uvac"  ./:mkcmd rje  (makes a single DQS11-based uvac system)
ARGS="dqsrje uvac2" ./:mkcmd rje  (makes uvac and uvac2)
ARGS="dqsrje uvac3" ./:mkcmd rje  (makes uvac, uvac2, and uvac3)
ARGS="kmcrje rje1"  ./:mkcmd rje  (makes a single KMC11-based IBM system)
ARGS="kmcrje rje2"  ./:mkcmd rje  (make rje1, and rje2)
ARGS="kmcrje rje3"  ./:mkcmd rje  (makes rje1, rje2, and rje3)
ARGS="kmcrje rje4"  ./:mkcmd rje  (makes rje1, rje2, rje3, and rje4)
```

2.8.1 UNIVAC Remote Job Entry

UNIVAC *rje* requires an additional tape to be used on the UNIVAC side of the system. The UNIVAC *rje* program is duplicated in the first and second files of its release tape in @COPY,G format. One should be read into a file called *RJEFIL*. The documentation on the configuration skeleton can be obtained by executing:

```
@DOC  RJEFILE.DOCELM
```

Examples of *rje* configurations are in the file elements *CONFIGRTN* (for level 35 or higher), and *CONFIGRTN/L32* and *CONFIGRTN/L33* (for levels 32 and 33 respectively). A start stream that will build the *rje* program, and cull it, is in *START/RJEMAK* (the account and project fields should be modified). A sample *rje* start stream where the absolute is placed in *RJE*RJE.RJECTL/PDP* is in *START/RJECTL*. Other sample runstreams using the file *SYS$*ABS.RJECTL/PDP* are in *RJEU1*, *RJEU2*, and *RJEU3* for using the first, second, or third configured line.

The LINEDATA cards in *CONFIGRTN* assume the EXEC is configured with 3 lines to a UNIX system whose CTM and LINE cards specify GB17, GB18, and GA01 respectively, and whose STATION cards specify a LOCAL station of UNIX01, UNIX02, and UNIX03 respectively. The LINEDATA cards in *CONFIGRTN/L32* and *CONFIGRTN/L33* assume the EXEC is configured with one line to a UNIX system whose LTG ID is 'UNXRJE' and whose REMOTE TERMINAL card specifies BPDP01.

3. CONFIGURATION PLANNING

3.1 UNIX Configuration

The basic UNIX operating system supplied supports only the console, a disk controller (disk drive 0), and a tape controller (tape drive 0). The actual configuration of your system must be described by you. All of the UNIX operating system source code and object libraries are in */usr/src/uts*. All of the configuration information is kept in the directory */usr/src/uts/*/cf*, the "*" represents either *pdp11* or *vax*. There are only two files that must be changed to reflect your system configuration, *low.s* (*univec.c* on the VAX) and *conf.c*; the program *config*(1M) should be used to make these changes.

Config requires a *system description file* and produces the two needed files. The first part of the system description file lists all of the hardware devices on your system. Next, various system information is listed. A brief explanation of this information follows; for more details of syntax and structure, see *config*(1M) and the associated *master*(5); TABLE I lists the values and sizes of the various parameters for the different CPUs.

TABLE I. UNIX Configuration Parameters

Item	PDP-11/34, /23		PDP-11/45, /70		VAX-11/780	
	Value	*Size*	*Value*	*Size*	*Value*	*Size*
nswap	1000	–	3000	–	9000	–
buffers	15-20	26†	25-60	26†	50-120‡	560
sabufs	4-6	538	10-15	538	–	–
inodes	30-50	74	100-250	74	100-250	76
files	30-50	8	100-250	8	100-250	12
mounts	3-5	8	8-16	8	8-16	16
coremap	50-100	4	50-100	4	–	–
swapmap	50-100	4	50-100	4	50-100	4
calls	15-30	6	30-60	6	30-60	12
procs	30-50	30	50-200	30	50-200	52
texts	10-15	12	25-50	12	25-50	16
clists	10-20	28	100-300	28	100-300	32
maxproc	15	–	15	–	25	–

† Plus 512 bytes outside system space.
‡ 127 buffers is the system maximum.

- *root*– Specifies the device where the *root* file system is to be found. The device must be a block device with read/write capability because this device will be mounted read/write as "/". Thus, a tape can not be mounted as the *root*, but can be mounted as some read-only file system. Normally, *root* is disk drive 0, section 0.
- *pipe*– Specifies where pipes are to be allocated (must be a *mounted* file system– the root file system is normally used).
- *dump*– Specifies the device to be used to dump memory after a system crash. Currently only the TU10 and TU16/TE16 tape drives are supported for this purpose.
- *swap*– Specifies the device and blocks that will be used for *swapping*. *Swplo* is the first block number used and *nswap* indicates how many blocks, starting at *swplo*, to use. Care must be taken that the swap area specified does not overlap any file system. For example, if section 0 is 8,000 blocks long, the *root* file system could occupy the first 6,000 blocks and *swap* the remaining 2,000 by specifying:

```
root rp06 0
swap rp06 0 6000 2000
```

- *buffers*– Specifies how many *system buffers* to allocate. Real time response improves as more buffers are allocated. UNIX buffers form a "data cache". Improvement in the hit rate of this cache tends to fall as the number of buffers is increased.
- *sabufs*– PDP-11 only: specifies how many *system addressable* buffers to allocate. One buffer is needed for every mounted file system. Certain I/O drivers need such buffers.
- *inodes*– Specifies how many *inode table* entries to allocate. Each entry represents a unique open inode. When the table overflows, the warning message "Inode table overflow" will be printed on the console. The table size should be increased if this happens regularly. The number of entries used depends on the number of active processes, texts, and mounts.
- *files*– Specifies how many *open-file table* entries to allocate. Each entry represents an open file. When the table overflows, the warning message "no file" will be printed on the console. The table size should be increased if this happens regularly.
- *mounts*– Specifies how many *mount table* entries to allocate. Each entry represents a

mounted file system. The *root* (/) will always be the first entry. When full, the *mount*(2) system call will return the error EBUSY.

- *coremap*— PDP-11 only: specifies how many entries to allocate to the *list of free memory*. Each entry represents a contiguous group of 64-byte blocks of free memory. When the list overflows, due to excessive fragmentation, the system will undoubtedly crash in an unpredictable manner. The number of entries used depends on the number of processes active, their sizes, and the amount of memory available.
- *swapmap*— Specifies how many entries to allocate to the *list of free swap blocks*. Exactly like the *coremap*, except it represents free blocks in the swap area, in 512-byte units.
- *calls*— Specifies how many *call-out table* entries to allocate. Each entry represents a function to be invoked at a later time by the clock handler. The time unit is 1/60 of a second. The call-out table is used by the terminal handlers to provide terminal delays and by various other I/O handlers. When the table overflows, the system will crash and print the panic message "Timeout table overflow" on the console.
- *procs*— Specifies how many *process table* entries to allocate. Each entry represents an active process. The scheduler is always the first entry and *init*(8) is always the second entry. The number of entries depends on the number of terminal lines available and the number of processes spawned by each user. The average number of processes per user is in the range of 2-5. When full, the *fork*(2) system call will return the error EAGAIN.
- *texts*— Specifies how many *text table* entries to allocate. Each entry represents an active read-only text segment. Such programs are created by using the −i or −n option of the loader *ld*(1). The −n option is implicit on the VAX. When the table overflows, the warning message "out of text" is printed on the console.
- *clists*— Specifies how many *character list buffers* to allocate. Each buffer contains up to 24 bytes. The buffers are dynamically linked together to form input and output queues for the terminal lines and various other slow-speed devices. The average number of buffers needed per terminal line is in the range of 5-10. When full, input characters from terminals will be lost and not echoed.
- *maxproc*— Specifies how many concurrent processes a non-super-user is allowed to run.
- *power*— Specifies whether to attempt restart after a power failure. A value 0 (default) indicates no restart, a value of 1 attempts power-fail restart. On restart, device drivers are called and process 1 (i.e., *init*) is sent a hangup signal; see *init*(8). VAX power-fail is provided for experimental use only in UNIX 3.0.

3.2 UNIX Generation

Before generating your *first* UNIX system, you must modify the file called *Makefile* in the */usr/src/uts/**/*cf* directory. This file contains four symbols that are used for system identification; it is used to initialize the internal *utsname* structure (see *uname*(1), *uname*(2), and *utsname*(5)). The four symbols (which are 8 characters maximum) are as follows:

SYS	your system name (e.g., *pwba*);
NODE	the name by which your system is known on the *uucp*(1) network (e.g., *pwba*);
REL	the operating system release (e.g., *UNIX3.0*);
VER	the current version of the system; this is usually four characters indicating when the system was made (e.g., *0620* for June 20).

Generally, only the first two symbols need local modification; the REL symbol is a constant for the duration of this release, while the VER symbol will be defined when you *make*(1) the system. The name of the executable file produced by the generation procedure will be the concatenation of the SYS and VER symbols (e.g., *pwba0620*).

To generate a new UNIX operating system, follow the procedure below:

```
cd /usr/src/uts/*/cf
ed dfile
a
    [information as described above]
.
w
q
config dfile
make VER= ver
```

The PDP-11 system has a relatively small address space, so that if table sizes or the number of device types are too large, various error messages will result and the above procedure will only create an *a.out* file. In particular, the maximum available data space is 49,152 bytes (57,344 bytes on the PDP-11/23 and the /34). The actual data space requested can be found by using *size*(1) on *a.out* and adding the *data*, *bss*, and, for PDP-11/23 and /34, text segment sizes. One then reduces the specified values for the various system entries until it all fits. The amount of space in the *bss* segment used for each entry is indicated in Section 3.1 above.

When you are satisfied with the new system, you can test it by the following procedure:

```
cp /usr/src/uts/*/sysver /          # sysver as above
cd /
rm /unix
ln /sysver /unix
sync
```

Halt the processor and reboot the system. Note that this procedure results in two names for the operating system object, the generic */unix*, and the actual name, say */utsa0501*. An old system may be booted by referring to the actual name, but remember that many programs use the generic name */unix* to obtain the *name-list* of the system.

If the new system does not work, verify that the correct device addresses and interrupt vectors have been specified. If the *wrong* interrupt vector and the *correct* device address have been specified for a device, the operating system will print the error message "stray interrupt at XXX" when the device is accessed, where XXX is the correct interrupt vector. If the *wrong* device address is specified, the system will crash with a panic trap of type 0 (indicating a timeout) when the device is accessed.

3.3 Special Files

A special file must be made for every device on your system. Normally, all special files are located in the directory */dev*. Initially, this directory will contain:

console	console terminal
error	see *err*(4)
mem, kmem, null	see *mem*(4)
tty	see *tty*(4)
rp[0-7], rrp[0-7]	disk drive 0, sections 0-7
rl[0-1], rrl[0-1]	disk drives 0 and 1
rk[0-1], rrk[0-1]	disk drives 0 and 1
mt0, rmt0	tape drive 0 (800 bpi)
mt4, rmt4	tape drive 0 (800 bpi, no rewind).

These special files are of two types, block and character. This is indicated by the character *b* or *c* in the listing produced by *ls*(1) with the −l option.

In addition, each special file has a major device number and a minor device number. The major device number refers to the device type and is used as an index into either the *bdevsw* or *cdevsw* table in the configuration file *conf.c*. The minor device number refers to a particular

unit of the device type and is used only by the driver for that type. The *config* program with the −t option will list major device numbers.

The program *mknod*(1M) creates special files. For example, the following would create *part* of the initially-supplied */dev* directory (on the PDP-11):

```
cd /dev
mknod console c 0 0
mknod error c 20 0
mknod mem c 2 0; mknod kmem c 2 1; mknod null c 2 2
mknod tty c 13 0
mknod rp0 b 0 0; mknod rrp0 c 7 0
mknod mt0 b 1 0; mknod rmt0 c 6 0
mknod mt4 b 1 4; mknod rmt4 c 6 4
```

After the special files have been made, their access modes should be changed to appropriate values by *chmod*(1). For example:

```
cd /dev
chmod 622 console
chmod 444 error
chmod 440 mem kmem
chmod 666 null
chmod 666 tty
chmod 400 rp0 rrp0
chmod 666 mt0 rmt0
chmod 666 mt4 rmt4
```

Minor device numbers for tape are a bit peculiar; the minor device number consists of the following four bits:

0= 800-bpi 1= 1600-bpi	0= rewind 1= no rewind	drive	select

Therefore, the special file for tape drive 1, for operation at 1600-bpi, with no rewind on close would have a minor device number of 13.

Note that file names have no meaning to the *operating system* itself; only the major and minor device numbers are important. However, many *programs* expect that a particular file is a certain device. Thus, by convention, special files are named as follows:

block device	*conf.c*	*/dev*
RP03 disk	rp	rp•
RP04/5/6 disk	hp	rp•
RS03/4 fixed head disk	hs	rs•
RK05 disk	rk	rk•
RL01 disk	rl	rl•
TU10 tape	tm	mt•
TE/TU16 tape	ht	mt•

character device	*conf.c*	*/dev*
DL11 asynch. line	kl	tty•, console
DH11 asynch. line mux	dh	tty•
DMC11 sync. unit	dmc	dmc•
DZ11 asynch. line mux	dz	tty•
DN11 auto call unit	dn	dn•
DU11 synch. line	du	du•
KMC11 micro	kmc	kmc•
DZ11/KMC11 assist	dza,dzb	tty•
LP11 line printer	lp	lp•
RP03 disk	rp	rrp•
RP04/5/6 disk	hp	rrp•
RS03/4 fixed head disk	hs	rrs•
TU10 tape	tm	rmt•
TE/TU16 tape	ht	rmt•
error	err	error
memory	mm	mem, kmem, null
terminal	sy	tty

For those devices with a */dev* name ending in "•", this character is replaced by a string of digits representing the *minor* device number. For example:

```
mknod /dev/mt1 b 1 1
mknod /dev/rp24 b 0 024
mknod /dev/tty03 c 1 3
```

Note that for disks, an octal number scheme is maintained because each drive is split eight ways. Thus, */dev/rp24* refers to section 4 of physical drive 2. There is also a special file, */dev/swap*, that is used by the program *ps*(1). This file must reflect what device is used for swapping and must be readable. For example:

```
rm /dev/swap
mknod /dev/swap b 0 0
chmod 440 /dev/swap
chown sys /dev/swap; chgrp sys /dev/swap
```

3.4 File Systems

Each physical pack is split into eight logical sections. This split is defined in the operating system by a table with eight entries. Each table entry is two words long. The first specifies how many blocks are in the section, the second specifies the starting cylinder; see *hp*(4) (RP04/5/6) and *rp*(4) (RP03) for default cylinder and block assignments. These values are described to the system in the header file */usr/include/sys/io.h*.

A file system starts at block 0 of a section of the disk and may be as large as the size of that section; if it is smaller than the size of a section, the remainder of that section is unused. Note that the sections themselves may overlap physical areas of the pack, but the file systems must *never* overlap.

The program *mkfs*(1M) is used to initialize a section of the disk to be a file system. Next, the program *labelit*(1M) is used to label the file system with its name and the name of the pack. Finally, the file system may be checked for consistency by using *fsck*(1M). The file system may then be mounted using *mount*(1M).

3.5 DZ11 software with KMC11 assist

KMC microprocessors may be used to control DZ11 asynchronous multiplexers, thus off-loading terminal-oriented functions from the main CPU. The software is distributed in two forms. The KMC11-A version does DMA output of data, character translations, tab expansions, etc. The

KMC11-B version does these output functions in addition to doing DMA input of data. Each KMC11 can control up to four DZ11 multiplexers for a total of thirty-two asynchronous lines. Each system can support up to four KMC11 microprocessors. Up to sixty-four DZ11 lines can be controlled by KMC11 microprocessors.

3.5.1 Installation

1. Generate a system (see Section 3.2 above) by including each DZ11 to be controlled by a KMC11 in the configuration file. For example:

```
#   For the KMC11-A
dza X Y Z

#   For the KMC11-B
dzb X Y Z
```

   where X is the vector address, Y is the UNIBUS address, and Z is the bus request level. Also include the KMC11 microprocessors in the configuration file:

```
kmc X Y Z
```

2. Install the KMC11 microcode in */lib*:

```
#   For the KMC11-A
cd /usr/src/uts/*/up/dza
/lib/cpp dza.s | kas -o /lib/dzkmc.o

#   For the KMC11-B
cd /usr/src/uts/*/up/dzb
/lib/cpp main.s | kasb -o /lib/dzkmc.o
```

3. Copy *dzkload* into */etc*:

```
#   For the KMC11-A
cp /usr/src/uts/*/up/dza/dzkload /etc

#   For the KMC11-B
cp /usr/src/uts/*/up/dzb/dzkload /etc
```

4. Update */etc/rc* to execute *dzkload* for multi-user and power-fail *init* states. Each KMC11 used to control a DZ11 must be loaded with microcode. For each KMC11 used to control a DZ11 include:

```
/etc/dzkload /dev/kmc?
```

   where ? is the minor device number of that KMC11.

5. Special files (see Section 3.3 above) must be created for each KMC11 and DZ11 line:

```
/etc/mknod /dev/kmc? c X ?
/etc/mknod /dev/tty?? c Y Z
```

   X is the major device number of the KMC11 and ? is the minor device number of the KMC11 controlling the DZ11 multiplexers, i.e., the KMC11 loaded with microcode in step 4. Y is the major device number of the *dza/dzb* device as is supplied by *config*(1M). Z is the minor device number for a particular line on a DZ11. This number is composed of three fields. The low-order three bits are the line number relative to a DZ11. The next three bits contain the minor device number of the DZ11 controlling these lines. Note that this number is the absolute DZ11 number, not the number relative to the KMC11. The high-order two bits are the minor device number of the KMC11 controlling this DZ11. For example:

mknod /dev/tty?? c Y 0241

specifies the second line (0 through 7) on the fifth DZ11 to be controlled by the KMC11 with minor device number 2. The DZ11 number is specified by the order of appearance in the configuration file. The first four DZ11 multiplexers must be associated with one KMC11 and the next four must be associated with another KMC11. The order in which the KMC11 microprocessors are specified is not significant.

4. ADMINISTRATIVE FILES

4.1 /etc/motd

This file contains the *message-of-the-day*. It is printed by */etc/profile* after every successful *login*.

4.2 /etc/rc

On the transition between init states, */etc/init* invokes */bin/sh* to run */etc/rc* (must have executable modes). So that */etc/rc* can properly handle the removal of temporary files and the mounting and unmounting of file systems, it is invoked with three arguments: new state, number of times this state has been entered, previous state. When the system is initially booted, */etc/rc* is invoked with arguments "1 0 0"; when state two (multi-user) is subsequently entered, the arguments are "2 0 1".

When state 2 is entered more than once (via another /etc/init 2), no action will be taken by this file. This enables the system administrator to add or delete *login* processes while the system is in multi-user mode (see */etc/inittab* below).

Daemons may be invoked either by */etc/rc* or by including lines for them in */etc/inittab*.

The */etc/rc* file is also used to initialize KMC11 microprocessors (see */etc/dzkload* below).

This file must be edited to suit local conditions; see *init*(8).

4.3 /etc/inittab

This file indicates to */etc/init* which processes to create in each *init* state. By convention, state 1 is *single-user* and state 2 is *multi-user*. For example, the following line creates a sample single-user environment:

```
1:co:c:/bin/sh </dev/console > /dev/console 2>&1
```

This indicates that for state 1, a process with the arbitrary unique identifier *co* should be created. The program invoked for this process should be the shell and when it exits it should be reinvoked (c flag).

To attach a *login* process to the console in the multi-user state, add the line:

```
2:co:c:/etc/getty console 4
```

and for line */dev/tty00* for use by 300/150/110 baud terminals, add the following line:

```
2:00:c:/etc/getty tty00 0 3600
```

The arguments to *getty* are the device, speed table, and number of seconds to allow before hanging up the line.

Again, this file must be edited for local conditions; see *getty*(8), *init*(8), and *inittab*(5).

4.4 /etc/dzkload

This file is invoked as a command by */etc/rc*. It contains instructions for initializing a KMC11 microprocessor that is to function as a controller for one or more DZ11 communications multiplexers (see Section 3.5 above). This file must be edited to suit the configuration.

4.5 /etc/passwd

This file is used to describe each *user* to the system. You must add a new line for each new user. Each line has seven fields separated by colons:

1. Login name: normally 1-6 characters, first character alphabetic, the remainder alphanumeric, no upper-case characters.
2. Encrypted password: initially null, filled in by *passwd*(1). The encrypted password contains 13 bytes, while the actual password is limited to a maximum of 8 bytes. The encrypted password may be followed by a comma and up to 4 more bytes of password "age" information.
3. User ID: a number between 0 and 65,535; 0 indicates the super-user. User IDs 0-99 are reserved.
4. Group ID: the default is group 1 (one). Group IDs 0-99 are reserved.
5. Accounting information: this field is used by various accounting programs. It usually contains the user name, department number, and account number.
6. Login directory: full path-name (keep them reasonably short).
7. Program name: if null, */bin/sh* is invoked after a successful login. If present, the named program is invoked in place of */bin/sh*.

For example:

```
ghh::138:1:6824-G.H.Hurtz(4357):/usr/ghh:
grk::244:1:6510-S.P.LeName(4466):/usr/grk:/bin/rsh
```

See also *passwd*(5), *login*(1), *passwd*(1).

4.6 /etc/group

This file is used to describe each *group* to the system. You must add a new line for each new group. Each line has four fields separated by colons:

1. Group name: normally 1-6 characters, first character alphabetic, rest alphanumeric, no upper-case characters.
2. Encrypted password: contains 13 bytes, while the actual password is limited to a maximum of 8 bytes.
3. Group ID: a number between 0 and 65,535. Group IDs 0-99 are reserved.
4. Login names: list of all *login* names in the group, separated by commas.

We strongly discourage group passwords. See also *group*(5).

4.7 /etc/profile

When the shell is executed and is the leader of a process group, as is the case when it is invoked by *login*, it will read and execute the commands in */etc/profile* before executing commands in the user's *.profile* file. This allows the system administrator to set up a standard environment for all users (e.g., executing *umask*, setting shell variables, etc.) and take care of other housekeeping details (such as *news* $-n$).

4.8 /etc/checklist

This file contains a list of default devices to be checked for consistency by the *fsck*(1M) program. The devices normally correspond to those mounted when the system is in *multi-user* mode. For example, a sample checklist would be:

```
/dev/rp0
/dev/rrp1
```

Note that the *root* device is specified as a *block* device, while all others are specified as *character* devices. Character devices can be checked faster than block devices. The *root* device is specified as a block device in order for the *fsck* program to detect when the *root* is being

checked, so that any modifications to this file system will result in an immediate reboot request.

4.9 /etc/shutdown

This file contains procedures to gracefully shut down the system in preparation for file-save or scheduled down-time.

4.10 /etc/filesave.?

This file contains the detailed procedures for the local file-save.

4.11 /usr/adm/pacct

This file contains the process accounting information; see *acct*(1M).

4.12 /usr/adm/wtmp

This file is the log of *login* processes.

5. REGENERATING SYSTEM SOFTWARE

System source is issued under the directory */usr/src*. The sub-directories are named *cmd* (commands), *lib* (libraries), *uts* (the operating system), *head* (header files), and *stand* (stand-alone programs); see *mk*(8) for details on how to remake system software.

A couple of anomalies: the accounting routine that deals with holidays and the prime/non-prime time split must be recompiled at the end of each year (it is currently correct for BTL-Murray Hill in 1980). The file is */usr/src/cmd/acct/lib/pnpsplit.c*. A reminder is sent to */usr/adm/acct/nitelog*, the standard place for such messages, starting a week before year-end and continuing until *pnpsplit.c* is recompiled.

6. FILE SYSTEM CONVERSION TO UNIX (VAX)

Procedures have been developed for converting UNIX file systems from PDP-11/UNIX (including UNIX/TS, PWB/UNIX Release 2.0, and Research Version 7) to VAX UNIX. Direct conversion from other systems (i.e., Version 6-based or UNIX/RT) is also possible, but the administrator should get help.

The following UNIX commands are referenced in this section: *cpio*(1), *find*(1), *fsck*(1M), *fscv*(1M), *mkdir*(1), *mkfs*(1M), *mount*(1M), and *umount*(1M). The reader is assumed to be familiar with them.

Unless you have repealed Murphy's Law, you should allow plenty of time for the conversion. As a lower bound, it takes about two hours to convert each 65K of file system space.

6.1 Preliminaries

Obviously, the new system should be generated and decently tested before conversion is attempted. Source for local commands and libraries should be moved to the VAX and compiled and tested. Your users may also reasonably require time to develop conversion programs for data files that contain binary information.

6.1.1 The Old System

The file systems should be pruned of marginal files. The following shell sequence will get rid of all executables:

```
#   For each user file system:
find /usr-file-system -type f -print | xargs file | \
    sed -n -e 's/\([^:]*\):.*executable/\1/p' >/usr/tmp/exfiles
#   You may want to look this file over before the next step.
xargs rm -f</usr/tmp/exfiles
rm /usr/tmp/exfiles
```

6.1.2 Spare Packs

Do not convert without spare packs—that is courting disaster. It is best to keep the old packs for several days, and to make backup tapes as well.

6.2 Converting the Hard Way

Using *find* and *cpio* takes much longer, but you will have optimized converted user file systems when you are finished (compacted *inodes* and directories, file and free-list blocks arranged for fastest access).

6.2.1 Copying from the Old System

The following steps should be executed by the super-user on an idle, stand-alone (old) system:

```
#   For each user file system:
cd /file-system-name
find . -cpio /dev/rmt1
```

The **-cpio** option of *find* produces ten-block records on physical tape in *cpio* format. Unless there are a great many linked files, a 1,600-bpi, 2,400-foot reel should hold about 65K file system blocks. If you have larger file systems, the easiest (fastest, safest) thing to do is to use a raw disk pack in place of the tape (i.e., */dev/rrp??* in place of the */dev/rmt1* above). In our tests, multi-reel *find/cpio* tapes have worked. *Find* can also be used to pick up *parts* of file systems that can be combined later as described below.

6.2.2 Under the New System

Re-create each file system as follows:

```
mkdir /file-system-name
mkfs /dev/rrp? blocks:inodes 7 418
labelit /dev/rrp? file-system-name pkname
mount /dev/rp? /file-system-name
cd /file-system-name
#   Mount tape created during step 3
cpio -idmB </dev/rmt1
#   If you are combining the smaller file systems,
#   you may copy-in more than one tape per new file system
#   (but make sure that first-level directory names are unique)
```

After the tapes have been copied in, the new file systems should be unmounted and checked (using *fsck*(1M)).

6.3 Converting the Easy Way

The *fscv*(1M) command has been provided for fast conversion between PDP-11 and VAX file systems.

Note that *fscv* will not convert "special files" (user file systems only). *Fscv* source will compile and run on either system. It was designed primarily for use in computer labs where there are a mixture of PDP-11 and VAX systems.

Example 1:

```
#  Converting PDP-11 to VAX:
#  Make sure that file system cylinder boundaries agree!
fscv -v /dev/rrp21 /dev/rrp31
```

Example 2:

```
#  Converting "in place" to the PDP-11:
#  Anyone who does this without making a copy first deserves
#  whatever bad (plenty) that can happen!
fscv -p /dev/rrp12 /dev/rrp12
```

6.4 A Final Precaution

It is only sensible to do another complete file system backup under the new operating system (using another set of tapes or packs).

ATTACHMENT 1

Processors and Peripherals Supported by UNIX

Processor	PDP-11/23	PDP-11/34	PDP-11/45	PDP-11/70	VAX-11/780
Memory	256KB ECC MOS	256KB ECC MOS	256KB ECC MOS or core	0.5-1MB ECC MOS or core	1-4MB ECC MOS
Disk drives:					
RL01 (2 required)	F	F	S	N	–
RK05 (2 required)	–	O	O	–	–
RP03	–	O	O	–	–
RP04	–	O (UNIBUS)	O (UNIBUS)	O (MASSBUS)	–
RP06	–	S (UNIBUS)	F (UNIBUS)	F (MASSBUS)	F (MASSBUS)
Fixed-head disks:					
RF11	–	–	O	–	–
RS03,4	–	–	N (UNIBUS)	N (MASSBUS)	–
Tape drives:					
TU10	–	O	O	–	–
TU16	–	O	O	O	–
TE16	–	F	F	F	F
TU45	–	–	–	S	S
Line printer:					
LP11†	–	N	N	N	N
Asynch. interfaces:					
DLV11	S	–	–	–	–
DL11E	–	S	S	S	–
DZ11	–	S	S	S	S
KMC11B/DZ11‡	–	F	F	F	F
DH11	–	O	O	O	–
Synch. interfaces:					
KMC11B/DMC line unit	–	S	S	S	S
DMC11	–	S	S	S	S
DU11	–	O	O	O	–
DQS11B/A	–	–	O	O	–
Auto-call unit:					
DN11AA/DA	–	S	S	S	S
Parallel interface:					
DRV11	S	–	–	–	–
DR11C/B	–	S	S	S	–
DA11B (use DMCs)	–	S	S	S	–
PCL11B	–	S	S	S	S

Key: F → First choice.
S → Supported.
O → Driver provided, but device is obsolete.
N → Driver provided, but device is not recommended.
– → No support.

† Use a printer on an RS232 interface.

‡ 4 DZ11s (32 lines) per KMC.

Administrative Advice for UNIX/TS

R. C. Haight

Bell Laboratories
Murray Hill, New Jersey 07974

The material presented here is based on the author's experiences and opinions. Nevertheless, it may prove useful. The material on phototypesetting was contributed by D. W. Smith.

1. ADMINISTRATOR'S ROAD MAP

Getting started as a UNIX†/TS system administrator is hard work. There are no real shortcuts to a working knowledge of the system. You will need time for reading, study and hands-on experimenting. Don't commit yourself to "going live" with your system until you have had two weeks to teach yourself your job, and get the initial hardware quirks ironed-out.

Don't consign the *Setting Up UNIX/TS* document to oblivion after your initial system "gen". In addition to needing it again whenever you add/change equipment, you will find that it contains valuable material about system tuning (buffers, *clists*, etc.) that appears nowhere else.

As an administrator, you should be familiar with a lot of the distributed documentation. The *Internals, Operations, and Administration* papers from *Documents for UNIX/TS* should all be studied, as well as the *Introduction*, *How to Get Started*, and most of the entries of the *UNIX/TS User's Manual*. In that manual, you should pay special attention to: *acct*(1M), *chmod*(1), *chown*(1), *config*(1M), *cpio*(1), *date*(1), *df*(1), *du*(1), *ed*(1), *env*(1), *find*(1), *fsck*(1M), *kill*(1), *mail*(1), *mkdir*(1), *mkfs*(1M), *ncheck*(1M), *ps*(1), *rm*(1), *rmdir*(1), *shutdown*(1M), *stty*(1), *su*(1), *sync*(1M), *time*(1), *volcopy*(1M), *wall*(1M), *who*(1), and *write*(1) in Section 1; all of Section 4; *acct*(5) in Section 5; and *crash*(8) and *vaxops*(8) in Section 8.

2. SYSTEM CAPACITY

The figures below are approximations based on our experience over several years:

Hardware Configuration	*Number of Simultaneous Users*
PDP-11/23; 256K-byte memory; 2 RL01 disks*	4
PDP-11/34; 256K-byte memory with cache; 2 RL01 disks*	8
PDP-11/45; 248K-byte memory; RP03 disk*	16
Above with RP06 (RP04, RP05) disk*	20
Above with memory cache	25
PDP-11/70; 512K-byte memory; RP06 (RP04, RP05) disks* (2 or more drives)	32
Above with 768K-byte memory and a disk drive (or fixed-head disk) set aside for the root file system	40
VAX-11/780; 1M-byte memory; at least 3 RP06 disks*	48

* Or equivalent.

See *Setting Up UNIX/TS* for the list of supported hardware options.

† UNIX is a Trademark of Bell Laboratories.

3. DISK FREE SPACE

Making files is easy under UNIX/TS, but if the free inode count falls below 100, the system spends most of its time rebuilding the free inode array. If a file system runs out of space (and the console switches are non-zero), the system prints "no-space" messages and does little else. To avoid problems, the following start-of-day free counts should be maintained:

- The file system containing */tmp* (temporary files):
  - 16-user system: 1,500 free blocks.
  - 40-user system: 3,000 free blocks.
- The file system containing */usr*:
  - 3,000 to 6,000 free blocks, depending on load.
- Other user file systems:
  - 6% to 10% free, depending on user habits (3,000 blocks minimum).

This brings up an associated problem: how big should file systems be? Our preference is to set aside space on each drive for a copy of root/swap and use the rest of the pack for a single file system. However, if you have user groups that fight over disk space, it may be better to split them up arbitrarily (i.e., divide a pack into more than one file system). Warning: if you set up different disk drives with differing cylinder partitions between file systems, it will probably lead to an operations goof someday.

4. A VERY FEW WORDS ABOUT SYSTEM TUNING

- As shipped, UNIX/TS has *no* programs with the text-bit mode set (see *chmod*(1)). The top contenders for the *t*-bit are *nroff* and *troff* followed (generally) by the larger phases of the C compiler (including the assembler and loader). The *t*-bit is only meaningful with pure text programs (*ld*(1) options *-i* or *-n*). Don't overdo it.
- File system reorganization (described below) can help throughput, but at the expense of down time. If you do it when your users are all asleep, it can help.
- If you use normal *shutdown* and */filesave.u* procedures, the file system check program (*fsck*(1M), *-S* option) will help keep the disk free list in reasonable order.
- Try to keep disk drive usage balanced. If you have over 20 users, the *root* file system (*/tmp*, */tmp*, */etc*, and *swap*) deserves a drive of its own.
- If you have a noisy modem (poorly executed do-it-yourself null-modem) or a disconnected modem cable, UNIX/TS will spend a lot of time trying to get it logged in. A random check of systems uncovers a lot of this going on.

5. WHY YOU MUST HAVE A SPARE DISK DRIVE

- Without a spare disk drive, the system will be *down* when a drive is *down*.
- Without a spare drive, it is difficult to reorganize file systems and to restore user files.

6. DISK PACKS

- Buy only fully ECC correctable packs and test them.
- If a pack develops uncorrectable errors, recondition it, or get rid of it.

RP06 disk packs used with UNIX/TS need not be totally error-free, but must be "flag-free". The term flag-free means that there should be no unrecoverable ECC (Error Correcting Code) errors. Technically, proper ECC handling can recover from 11-bit error bursts. However, we hear that the length of bursts can grow as a pack ages. We recommend that no pack that has more than 3-bit error bursts be accepted. DEC has promised to revise their formatter to print this information in readable form. In the meantime, the following explanation may help (paraphrased from a DEC source).

In reading the formatter printout, ECC correctable errors are identified by the headings "DATA ERROR DURING WRITE CHECK." Error-register values are printed below the message. The two registers of interest are RPER1 and RPEC2. A RPER1 value of 1000000 indicates ECC (no other bits on). The RPEC2 register describes the bit span of the error. For example,

RPEC2=003774 means that there was an unacceptable 9-bit (binary 0000011111111100) error burst; RPEC2=000240 is an acceptable 3-bit span (0000000010100000—there may be zero bits mixed in). If such acceptable errors account for all "unrecoverable" errors reported (and there aren't too many of them), then you have a flag-free pack.

7. PROTECTING USER FILES

Users, especially inexperienced ones, occasionally remove their own files. Open files are sometimes lost when the system crashes. Once in a great while, an entire file system will be destroyed (picture a disk controller that goes bad and writes when it should read). Here is a suggested file backup procedure:

- Each day, copy all user file systems to backup packs. Keep these packs 3 to 5 days before re-using them.
- Once a week, copy each file system to tape. Keep weekly tapes for 8 weeks.
- Keep bi-monthly tapes "forever" (they should be re-copied once a year).

The most recent weekly tapes should be kept off premises. The other tapes should be in a fire-proof safe.

When UNIX/TS goes down, active files can get scrambled. Your users will not want to start the day over every time your system fails. In addition to good backup, you *must* have file-system patching expertise available (on-site or on-call). If you ever re-boot the system for general use without checking out the file systems, terrible things will happen (we once had five duplicate entries on a file-system free list—this ruined over 100 new files in just three days). Study *fsck*(1M) and *crash*(8).

8. UNIX/TS FILE SYSTEM BACKUP PROGRAMS

The following backup programs are distributed:

- *Dump/restor*: This is a familiar tape-based system that has been used for several years. Full dumps are usually taken when the *dump* program warns that an incremental dump will run to more than one reel.
- *Find/cpio*: UNIX/TS is distributed in *cpio* format. The *—cpio* option of the *find* command has made it time-competitive with *dump/restor*. However, it does not produce a "perfect" restore from a full dump plus incremental dump (new and changed files are OK, but file removal information is lost). Because of this, full dumps should be taken fairly often (weekly/bi-weekly). *Cpio* is the only program listed here that makes system-independent copies. It can be used to move files between various versions of MERT and UNIX, and can be used in system conversion.
- *Volcopy*: physical file system copying to disk or tape. For those who can afford a spare drive, *volcopy* to disk provides convenient file restore and quick recovery from disk disasters (remember the spare drive). Tape *volcopy* provides good long-term backup because the file system can be read-in fairly quickly, mounted, and browsed over. Disk and tape *volcopy* are generally used together for short- and long-term backup. *Volcopy* can also be used for full dumps with either *dump/restor* or *cpio/find*.

The table below summarizes attributes of these programs. The file system size is 65,500 blocks in all cases; times are in minutes; judgements are subjective.

	dump/restor	*find/cpio*	*volcopy (disk)*	*volcopy (tape)*
Full dump time	40	40	2	15
Incremental dump time	6	7	–	–
Full restore time	40(?)	80	2	15
Incremental restore time	8	10	–	–
Ease of restoring:				
one file	fair	fair	good	fair
a directory	poor	fair	good	good
scattered files	poor	poor	good	good
full restore	fair	fair	very good	good
Needs tape drive	yes	yes	no	yes
Needs spare file system (only when restoring)	no	no	–	yes
Needs spare disk drive (two CPUs can share)	–	–	yes	–
Maintains pack/tape labels	no	no	yes	yes
Handles multi-reel tape	yes	yes	–	yes
512 blocks per record	1,10	1,10	88	10
Interactive (i.e., ties up console)	no(?)	yes	yes	yes
May require separate I/D space	no	yes	no*	no

* Blocks per record are cut to 66 without separate I/D space.

We strongly recommend the spare disk drive: as explained in Section 5 above, the speed and convenience of *volcopy* are by no means the only advantage of a spare drive.

9. CONTROLLING DISK USAGE

If your UNIX/TS system is a success, you will soon run out of disk space:

- During the considerable delay before you can get more drives, you will need to control usage:
  - Try to maintain the start-of-day counts recommended above. Watch usage during the day by executing the *df* command regularly.
  - The *du*(1) command should be executed (after hours) regularly (e.g., weekly) and the output kept (in an accessible file) for later comparison. In this way you can spot users who are rapidly increasing their disk usage.
  - The *find*(1) can be used to locate inactive (or large) files. Example:

    find / –mtime +90 –atime +90 –print >somefile

    records in "somefile" the names of files neither written nor accessed in the last 90 days. Of course, this works best if you are super-user.
- You will also have to balance usage between file systems. To do this you will have to move user directories. Users should be taught to accept file system name changes (and to program around them—preferably ahead of time). The user's login directory name (available in the shell variable *HOME*) should be utilized to minimize path name dependencies. User groups with more extensive file system structures should set up a shell variable to refer to the file system name (e.g.: *FS*).
- The *find*(1) and *cpio*(1) commands can be used to move user directories and to manipulate the file system tree. The following sequence is useful (it moves, via magnetic tape, the directory trees *userx* and *usery* from file system *filesys1* to file system *filesys2*where, presumably, more space is available):

```
cd /filesys1
find userx usery -cpio /dev/rmt0
cd /filesys2
mkdir userx usery
chown userx userx
chown usery usery
cpio -idmB </dev/rmt0
#    Make sure new copy is OK
#    Change userx and usery login directories in the /etc/passwd file
rm -rf /filesys1/userx /filesys1/usery
```

When moving more than one user in this way:
- Keep users with common interests in the same file system (they may have linked files).
- Move groups of users who may have linked files with a single *cpio* (otherwise linked files will be unlinked and duplicated).

10. REORGANIZING FILE SYSTEMS

The procedure for moving users described above can be expanded to provide a way to reorganize whole file systems. Reorganization can improve system response time. This is particularly true of the *root* file system (which must be reorganized with all other file systems unmounted) and */usr*. Unfortunately, reorganization of large file systems is slow.

11. KEEPING DIRECTORY FILES SMALL

Directories larger than 5K bytes (320 entries) are very inefficient because of file system indirection. A UNIX/TS user once complained that it took the system ten minutes to complete the login process; it turned out that his login directory was 25K bytes long, and the login program spent that time fruitlessly looking for a non-existent *.profile* file. A large */usr/mail* directory can also really slow the system down. The following will ferret out such directories:

```
find / -type d -size +10 -print
```

Removing files from directories does not make the directories get smaller (the empty directory entries are available for reuse). The following will "compact" */usr/mail* (or any other directory):

```
mv /usr/mail /usr/omail
mkdir /usr/mail
chmod 777 /usr/mail
cd /usr/omail
find . -print | cpio -plm ../mail
cd ..
rm -rf omail
```

12. ADMINISTRATIVE USE OF "CRON"

The program *cron*(1M) is useful in the administration of the system; it can be used to:

- Turn off the programs in directory */usr/games* during prime time.
- Run programs off-hours:
  - accounting;
  - file system administration;
  - long-running, user-written shell procedures (using the *su*(1) command), for example:

```
su - userx userx_shell arg ...
```

13. WATCH OUT FOR FILES AND DIRECTORIES THAT GROW

- Accounting files:
  - */usr/adm/wtmp*—login information;

- */usr/adm/pacct*—process accounting; gets big quickly.

- Other files:
  - */usr/lib/cronlog*—status log of commands executed by *cron*(1M);
  - */usr/spool*—spooling directory for line printers, *uucp*(1C), etc., and whose sub-directories should be compacted as described above.

14. ALLOCATING RESOURCES TO USERS

A prospective user should obtain connect-time and file-space authorization through appropriate channels. Once this is done, the user should apply for a login by providing the following information to the "system administrator":

- User's name.
- Suggested login name (not more than 8 characters, beginning with a lower-case letter).
- Relationships to other users (this influences the choice of the file system).
- Estimate of required file space (this also influences the choice of the file system).

Users should be forced to have passwords (not more than 8 characters long, but more than 5, and *not* in Webster's Unabridged); *passwd*(5) explains how to do that.

15. THE MATTER OF ACCOUNTING AND USAGE

You should run the accounting programs even if you do not "bill" for service. Otherwise, your users' habits (especially *bad* habits) will be a mystery to you. Accounting information can help you find performance bottlenecks, unused logins, bad phone lines, etc.

16. DIAL LINE UTILIZATION

If prime-time dial line utilization gets much over 70%, users will start to encounter busy signals when dialing in. This, in turn, will lead to "line hogging". The only solutions are to get a larger (another) machine, or to get rid of users. Manual policing will help some, but "automatic" policing will be *invariably* subverted by users.

17. "BIRD-DOGGING"

When the system is busy (lines busy and/or slow response), someone should determine why this is so. The *who*(1) command lists the people logged in. The *ps*(1) command shows what they are doing. (The *letchwhodo* command combines the output of *who* and *ps*.) Unfortunately, *ps* operates from heuristics that can consistently fail to report certain processes in a busy system. That is, one must be careful about hanging up an apparently inactive line. The *accton*(1M) command can read the shell accounting file */usr/adm/pacct* backwards from the most recent entry. It will print entries for selected lines or login names.

18. 300/1,200-BAUD TERMINALS

Don't use upper-case-only terminals. Get full-duplex, full-ASCII terminals. Hardware horizontal tabbing is very desirable, because it increases output speed and lowers system overhead. A fair proportion of your terminals should provide for correspondence-quality hard-copy output to take advantage of the UNIX/TS word-processing capabilities; see *term*(7).

19. LINE PRINTERS

Most line printers are troublesome and impose considerable overhead on the system. Most also lack hardware tabs, character overstrike capability, etc. A printer that will work over an asynchronous link (DC1/DC3 protocol required) may be the best bet.

20. SECURITY

The current UNIX/TS is not tamper-proof. You can't keep people from "breaking" the system, but you can usually detect that they have done so. The following command will mail (to root)

a list of all "set user ID" programs owned by *root* (super-user):

```
find / -user root -perm -4100 -exec ls -l {} \; | mail root
```

Any surprises in *root*'s mail should be investigated. Related advice:

- Change the super-user password regularly. Don't pick obvious passwords (choose 6-to-8 character nonsense strings that combine alphabetics with digits or special characters).
- If you have dial ports and do not *require* passwords, you are courting trouble.
- The *chroot*(1M) ans *su*(1) commands are inherently dangerous, as are *group* passwords; consider removing them from "production" systems.
- Login directories, *.profile* files, and files in */bin*, */usr/bin*, */lbin*, and */etc* that are writable by others than their respective owners are security weak spots; police your system regularly against them.
- Remember, no time-sharing system with dial ports is really secure. Don't keep top-secret stuff on the system.

21. COMMUNICATING WITH YOUR USERS

The directory */usr/news* and the *news*(1) command are provided as a way to get *brief* announcements to your users. More pressing items (one-liners) can be entered in the */etc/motd* (message of the day) file; *motd* and (new to the user) *news* are announced at login time.

To reach users who are already logged in, use the *wall*(1M) (write all) command. Don't use *wall* while logged-in as super-user, except in emergencies.

The */usr/news* directory should be cleaned out every few weeks so that nothing older than, say, three months is ever found there. The *motd* file should be cleaned out daily.

We have found that, on most systems, a file in */usr/news* will reach 50% of the users within a day and over 80% of the users within a week.

22. TROUBLESHOOTING

It would be easy to write a book on this topic. The following are some of the key items:

a. Dealing with the hardware service contractor:

- Before you take out a hardware service contract (with DEC or with someone else), be sure that the contractor agrees to get along with the UNIX/TS software ("It's the hardware," says you; "It's the software," says the hardware service contractor).
- Keep on top of problems. For instance, DEC has a problem-aging priority scheme. Find out about any such scheme that your contractor may have, and make them prove that it is being followed. Remember that an unreported problem is getting no priority at all. If a problem persists, escalate it up your contractor's local management chain; it may also be effective to complain to your contractor's sales representative.
- If you are serious about service to your users, you should have an extended-period service contract (e.g., 16 hours/day, 6 days/week). Arrange for preventive maintenance, non-critical repair, and add-on installation work to be done before or after prime time.
- If you have a service contract, learn the details. In particular, make sure that preventive maintenance is scheduled in advance and that it is completed.
- Ask the hardware service contractor to provide and maintain a "site log". You will have to work on the log, as well.
- Make sure that your hardware vendor (as well as your hardware service contractor, if the two are different) agrees to the presence of non-DEC equipment on your system (even if you have none to start with).
- Run error logging. Keep console sheets. Make sure error messages are shown to your contractor's Customer Engineers.
- Take core dumps after system crashes and interpret results for Customer Engineers.
- Keep down-time records and make sure that your hardware service contractor knows about them.

b. Dealing with the telephone services vendor:

You are most apt to have telephone problems when you rearrange or add equipment. You may also have occasional central office, trunking, and modem failures:

- Be specific with repair operators: tell them that the trouble involves *data* equipment.
- If your first call fails to get results, ask for the "supervisor" on the second call, and, if necessary, escalate further to get the problem solved.

c. Some obvious problem areas:

- Disk Drives—Over 50% of your problems are likely to be related to the disk subsystem. As mentioned earlier, the way to keep your system *up* is to have a spare disk drive. Remember:
  - Preventive maintenance of disk drives is very important.
  - Make sure that the Customer Engineers who service your hardware see the error-logging printouts and console error messages produced by UNIX/TS (and that they understand them).
  - Disk failure can ruin a UNIX/TS file system. The *only* defense is to make a complete, daily file backup! (See *Protecting User Files* above.)
  - Many administrators believe the the RP04 disk drives fail more often than RP06s and take longer to fix.
- Dial Ports—In this area, as well as in the area of synchronous data interfaces, there is room for finger-pointing among all your vendors. Check for obvious things:
  - Is the system in "multi-user" mode?
  - Is the */etc/inittab* file OK?
  - Are any cables loose (*both* ends)?
  - In some telephone offices, trunk-hunting is based on 10-number groups. Hunting *between* such groups can fail independently of anything else.

  The possibilities for trouble are many. The "decision table" below attempts to describe some alternatives; it is meant primarily for users of DH11/DZ11 asynchronous devices. If you are unfamiliar with the format, (vertical) Rule 3 reads: "If line rings *and* ring light shows *and* computer does *not* answer *and* switching the modem solves the problem, then it is likely to be a telephone company problem: also, busy out that line."
- Early experience with the DZ11 has been poor. Several different problems have cropped up including bad line units and a stuck interrupt bit that crashes the system. Don't install DZs without giving them the full diagnostic treatment.
- Synchronous Ports—High-speed synchronous interface devices are even more trouble than dial equipment. The following is a list of potential trouble spots:
  - Your UNIX/TS software.
  - Your interface device (e.g., DQS11B).
  - Cable to your modem.
  - Your modem.
  - The communications line.
  - Other modem.
  - Other cable.
  - Other interface device.
  - Other system's software.

  Think of the finger-pointing possibilities. The best defense is a good line monitor.
- Power Supply Modules—There are a lot of them, and they do fail, more or less regularly. Hard failure can be detected at the console; voltage drift is tougher. Failure of the FP11 (floating-point unit) power supply can be slow to fix, because Customer Engineers are likely to work back from the far end of the "bus", taking a long time to find the problem.

Asynchronous Line Problems

Rules:	1	2	3	4	5	6	7	8	9	0
Condition:										
Line rings	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Ring light shows on telephone console	–	N	Y	Y	Y	Y	Y	Y	Y	Y
Computer answers	–	–	N	N	Y	Y	Y	Y	Y	Y
Login message received on terminal	–	–	–	–	N	N	Y	Y	Y	Y
Switching modem solves problem	–	–	Y	N	Y	N	–	–	–	–
User can login	–	–	–	–	–	–	N	N	N	Y
Telephone console shows data received	–	–	–	–	–	–	Y	Y	N	–
Problem affects whole DH/DZ (up to 16 lines)	–	–	–	–	–	–	Y	N	–	–
Diagnosis and/or Action:										
No problem	–	–	–	–	–	–	–	–	–	X
PDP-11 hardware problem likely	–	–	–	X	–	X	X	–	–	–
Telephone problem likely	X	X	X	–	X	–	–	–	X	–
May be a problem with user's terminal	–	–	–	–	–	–	–	X	–	–
Busy out bad line(s)	X	X	X	X	X	X	X	–	X	–

23. DATASET OPTIONS

The following dataset options seem to work with UNIX/TS:

The 801C-L1 (Auto-Call Unit):

Jumpers:

E2 to *E3*
E6 to *E5*

Options:

Y, X, T, B,
ZG, ZP, G,
R, ZT

Switches (0 = open, 1= closed, i.e., side next to number is down):

S1 = 1000[1] (Bracketed switches are missing on some models.)
S2 = 0101
S3 = 11010
S4 = 11[00]

The 212A-L1 (1,200-baud full-duplex):

Options:

E, ZF, YF, YC,
YG, YJ, YK,
S, V, A, T, ZH,
W, YP, YR

Switches:

S1 = [0]001
S2 = 110001000
S3 = 11110000
S5 = 00

24. NULL MODEM WIRING

Improperly wired null modems can cause spurious interrupts, especially at higher baud rates. A single bad modem on a 9,600-baud line can waste 15% of your CPU power. The following (symmetrical) wiring plan will prevent such problems:

pin 1 to 1
pin 2 to 3
pin 3 to 2
pins 4 and 5 to 8
pin 6 to 20
pin 7 to 7
pin 8 to 4 and 5
pin 20 to 6
ground all other pins to 7, SG.

25. 113D, 103J DATA SET PROBLEMS

The DH11 and DJ11 multiplexers normally have a jumper connecting pin 25 to pin 4 (request to send), thus asserting pin 25 when the line is opened. This jumper should be removed for any lines connected to 113Ds or 103Js (also applies to 103Js with 801s).

26. PHOTOTYPESETTING EQUIPMENT AND SUPPLIES

Read this section if you plan to use the phototypesetting software of UNIX/TS.

Phototypesetter. The phototypesetter and fonts currently supported by UNIX/TS are manufactured by:

Wang Graphic Systems, Inc.
Executive Drive
Hudson, NH 03051 (603-889-8550)

The phototypesetter is an on-line C/A/T System 1 with a high-speed turret. The external paper tape reader on the typesetter is not needed, because the typesetter is connected to the PDP-11 CPU via a DR11C.

PDP 11/45 Only. The following modification (developed by DEC Field Service) should be made to the DR11C (without this modification, the system may crash when the typesetter is powered down): "Add two 390-ohm resistors—from E-18 pin 6 to ground, and from E-18 pin 3 to ground. Put a piece of insulating tubing over the leads so that they do not short out the 'etch' runs that they cross."

Fonts. There are eight fonts that are normally used, as shown in the table below. The first three of the these provide the most-often used (serif) typeface. The last three are used when a sans-serif typeface is desired. The fourth font contains a number of Greek characters and mathematical symbols; see *NROFF/TROFF User's Manual* by J. F. Ossanna. The fifth font is useful for typesetting text that you wish to look like terminal or printer output, e.g., for examples of programs. Wang Graphic Systems, Inc. offers a variety of other fonts. For *troff* to be able to use these fonts, corresponding font tables must be built and compiled into the directory */usr/lib/font*.

Name	*Part Number*	*Troff Name*
BT Times Roman	802-016A	R
Times Italic	802-013A	I
Times Bold	802-014A	B
BT PI Font #4 Special Characters	829-021B	S
BT PI Font #6 Constant Width	829-046A	CW
Geneva (Helvetica) Regular	803-032B	G or H
Geneva (Helvetica) Regular Italic	803-033B	GI or HI
Geneva (Helvetica) Medium	803-034B	GM or HM

Other fonts for which the source font tables are supplied are:

Name	*Troff Name*
Boston Condensed	BC
News Condensed	C
Century Schoolbook Expanded	CE
Century Schoolbook Italic	CI
Century Bold Italic	CK
Century Schoolbook	CS
Futura (Utica) Demibold	FD or UD
Text Greek	GR
Geneva Light	L
Geneva Light Italic	LI
Palatino	PA
Palatino Bold	PB
Palatino Italic	PI
Stymie Bold	SB
Stymie Medium Italic	SI
Stymie Medium	SM

Paper and Chemicals. The phototypesetter "prints" onto photo-mechanical paper, which can be obtained from a photographic supply house and is specified as:

- Kodak Ektamatic Paper, Grade S, Type 2250,
  8 in. × 150 ft., Specification 175 (or equivalent).

Also obtainable from such a supply house are the chemicals for the developing process:

- Kodak Ektamatic A10 Activator (or equivalent).
- Kodak Ektamatic S40 Stabilizer (or equivalent).

These chemicals should be ordered in 9.5-liter (2.5-gallon) containers to work with the circulator.

Developer. A Kodak Ektamatic Processor Model 214K (or equivalent) is used to process the paper from the typesetter. A light-proof box attached to the 214K (to hold the output cassette from the typesetter) is called an "Autofeeder" and can be obtained from:

Peripheral Graphics, Inc.
Andover Industrial Center
York Street
Andover, MA 01810 (617-475-9005)

Circulator and Dryer. A circulator and a paper dryer, as well as a shelf for the dryer can be obtained from:

Mohr Enterprises
8015 North Ridgeway Ave.
Skokie, IL 60076 (312-674-8890)

The needed parts are:

- ME-8 Mohrflow: circulator to increase the usability of the chemicals.
- ME-5 Mohrdry: dryer for the photo-mechanical paper.
- Dryer Extension: shelf to support the dryer, it connects to the circulator cabinet.

Also obtainable from Mohr Enterprises are cleaners for the developer and circulator. Such cleaning is needed every 2 to 4 weeks, depending on the volume of work:

- R-53 Mohrchem Activator Cleaner Concentrate.
- R-57 Mohrchem Stabilizer Cleaner Concentrate.

Each quart bottle makes 9.5 liters (2.5 gallons) of reusable cleaner to clean the tubing, rollers, and tray of the developer and circulator. Equivalent cleaners can be obtained from other suppliers.

January 1980

The PWB/UNIX Accounting System

Henry S. McCreary

Bell Laboratories
Piscataway, New Jersey 08854

ABSTRACT

The PWB/UNIX* Accounting System provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C programs and shell procedures is provided to reduce this accounting data into summary files and reports. This memorandum describes the structure, implementation, and management of this accounting system.

* UNIX is a Trademark of Bell Laboratories.

The PWB/UNIX Accounting System

Henry S. McCreary

Bell Laboratories
Piscataway, New Jersey 08854

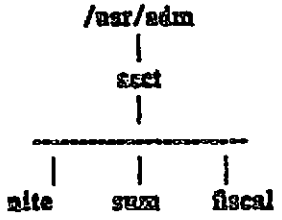
1. Introduction

The PWB/UNIX* accounting system was originally designed by John Mashey. Several modifications and additions have been made to make the system easier to manage, and to make it less susceptible to corrupted data or system errors. The following list is a synopsis of the actions of the accounting system:

- At process termination the UNIX Kernal writes one record per process in /usr/adm/pacct in the form of *acct.h*.[1]
- The *login* and *init* programs record connect sessions by writing records into /usr/adm/wtmp. Date changes, reboots, and shutdowns are also recorded in this file.
- The disk utilization program *acctdusg*, breaks down disk usage by login.
- Fees for file restores, etc, can be charged to specific logins with the *chargefee* shell procedure.
- Each day the *runacct* shell procedure is executed via *cron* to reduce accounting data, produce summary files and reports.[2]
- The *monacct* procedure can be executed on a monthly or fiscal period basis. It saves and restarts summary files, generates a report, and cleans up the sum directory. These saved summary files could be used to charge users for UNIX usage.

2. Files and Directories

The /usr/lib/acct directory contains all of the C programs and shell procedures necessary to run the accounting system. The adm login (UID 4) is used by the accounting system and has the following directory structure:



The /usr/adm directory contains the active data collection files.[3] The nite directory contains files that are re-used daily by the *runacct* procedure. The sum directory contains the cumulative summary files updated by *runacct*. The fiscal directory contains periodic summary files created by *monacct*.

---

* UNIX is a Trademark of Bell Laboratories.

1. See Attachment 2 for a description of data files.
2. See Attachment 3 for a sample report output.
3. For a complete explanation of the files used by the accounting system, see Attachment 1.

3. Daily Operation

When UNIX is switched into multi-user mode, /usr/lib/acct/startup is executed which does the following:

1. The *acctwtmp* program adds a "boot" record to /usr/adm/wtmp. This record is signified by using the system name as the login name in the wtmp record.
2. Process accounting is started via *turnacct*. *Turnacct on* executes the *accton* program with the argument /usr/adm/pacct.
3. The *remove* shell procedure is executed to cleanup the saved pacct and wtmp files left in the sum directory by *runacct*.

The *ckpacct* procedure is run via *cron* every hour of the day to check the size of /usr/adm/pacct. If the file grows past 1000 blocks (default), *turnacct switch* is executed. While *ckpacct* is not absolutely necessary, the advantage of having several smaller pacct files becomes apparent when trying to restart *runacct* after a failure processing these records.

The *chargefee* program can be used to bill users for file restores, etc. It adds records to /usr/adm/fee which are picked up and processed by the next execution of *runacct* and merged into the total accounting records.

Runacct is executed via *cron* each night. It processes the active accounting files, /usr/adm/pacct?, /usr/adm/wtmp, /usr/adm/acct/nite/disktacct, and /usr/adm/fee. It produces command summaries and usage summaries by login.

When the system is shut down using *shutdown*, the *shumacct* shell procedure is executed. It writes a shutdown reason record into /usr/adm/wtmp and turns process accounting off.

After the first re-boot each morning, the computer operator is instructed by */etc/rc* to execute /usr/lib/acct/prdaily to print the previous day's accounting report.

4. Setting up the Accounting System

In order to automate the operation of this accounting system, several things need to be done:

1. If not already present, add this line to the /etc/rc file in the state 2 section:
   /bin/su - adm -c /usr/lib/acct/startup
2. If not already present, add this line to */etc/shutdown* to turn off the accounting before the system is brought down:
   /usr/lib/acct/shutacct
3. Three entries should be made in /usr/lib/crontab so that *cron* will automatically start some shell procedures.
   0 4 * * 1-6 /bin/su - adm -c "/usr/lib/acct/runacct 1> /usr/adm/acct/nite/fd2log"
   0 2 * * 4 /bin/su - adm -c "/usr/lib/acct/ckdisk"
   5 * * * * /bin/su - adm -c "/usr/lib/acct/ckpacct"
4. The PATH shell variable in adm's .profile should be set to:
   PATH=/usr/lib/acct:/bin:/usr/bin

5. Runacct

Runacct is the main daily accounting shell procedure. It is normally initiated via *cron* during non-prime time hours. *Runacct* processes connect, fee, disk, and process accounting files. It also prepares daily and cumulative summary files for use by *prdaily* or for billing purposes. The following files produced by *runacct* are of particular interest.

nite/lineuse	Produced by *acctcon1*, which reads the wtmp file, and produces usage statistics for each terminal line on the system. This report is especially

useful for detecting bad lines. If the ratio between the number of logoffs to logins exceeds about 3/1, there is a good possibility that the line is failing.

nite/daytacct This file is the total accounting file for the previous day in *tacct.h* format.

sum/tacct This file is the accumulation of each day's nite/daytacct, which can be used for billing purposes. It is restarted each month or fiscal by the *monacct* procedure.

sum/dayems Produced by the *acctems* program, it contains the daily command summary. The ASCII version of this file is nite/dayems.

sum/cms The accumulation of each day's command summaries. It is restarted by the execution of *monacct*. The ASCII version is nite/cms.

sum/loginlog Produced by the *lastlogin* shell procedure, it maintains a record of the last time each login was used.

sum/rprt.*MMDD* Each execution of *runacct* saves a copy of the output of *prdaily*.

Runacct takes care not to damage files in the event of errors. A series of protection mechanisms are used that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that *runacct* can be restarted with minimal intervention. It records its progress by writing descriptive messages into the file active.[4] All diagnostic output during the execution of *runacct* is written into fd2log. To prevent multiple invocations, in the event of two crons or other problems, *runacct* will complain if the files lock and lock1 exist when invoked. The lastdate file contains the month and day *runacct* was last invoked, and is used to prevent more than one execution per day. If *runacct* detects an error, a message is written to /dev/console, mail is sent to root and adm, the locks are removed, diagnostic files are saved, and execution is terminated.

In order to allow *runacct* to be restartable, processing is broken down into separate reentrant states. This is accomplished by using a case statement inside an endless while loop. Each state is one case of the case statement. A file is used to remember the last state completed. When each state completes, statefile is updated to reflect the next state. In the next loop through the while, statefile is read and the case falls through to the next state. When *runacct* reaches the CLEANUP state, it removes the locks and terminates. *States* are executed in the following order:

SETUP The command turnacct switch is executed. The process accounting files, /usr/adm/pacct*?*, are moved to /usr/adm/Spacct*?.MMDD*. The /usr/adm/wtmp file is moved to /usr/adm/acct/nite/wtmp.*MMDD* with the current time added on the end.

WTMPFIX The wtmp file in the nite directory is checked for correctness by the *wtmpfix* program. Some date changes will cause *acctcon1* to fail, so *wtmpfix* attempts to adjust the time stamps in the wtmp file if a date change record appears.

CONNECT1 Connect session records are written to ctmp in the form of *ctmp.h*. The lineuse file is created, and the reboots file is created showing all of the boot records found in the wtmp file.

---

4. Files used by *runacct* are assumed to be in the nite directory unless otherwise noted.

CONNECT2	Ctmp is converted to ctacct.*MMDD* which are connect accounting records.[5]
PROCESS	The *acctpre1* and *acctpre2* programs are used to convert the process accounting files, /usr/adm/Spacct*?*.*MMDD*, into total accounting records in ptacct*?*.*MMDD*. The Spacct and ptacct files are correlated by number so that if *runacct* fails, the unnecessary reprocessing of Spacct files will not occur. One precaution should be noted; when restarting *runacct* in this state, remove the last ptacct file because it will not be complete.
MERGE	Merge the process accounting records with the connect accounting records to form daytacct.
FEES	merge in any ASCII *tacct* records from the file fee into daytacct.
DISK	On the day after the *sdisk* procedure runs, merge disktacct with daytacct.
MERGETACCT	Merge daytacct with sum/tacct, the cumulative total accounting file. Each day, daytacct is saved in sum/tacct*MMDD*, so that sum/tacct can be recreated in the event it becomes corrupted or lost.
CMS	Merge in today's command summary with the cumulative command summary file sum/cms. Produce ASCII and internal format command summary files.
USEREXIT	Any installation dependent (local) accounting programs can be included here.
CLEANUP	Clean up temporary files, run *prdaily* and save its output in sum/rprt*MMDD*, remove the locks, then exit.

6. Recovering from failure

The *runacct* procedure can fail for a variety of reasons; usually due to a system crash, /usr running out of space, or a corrupted wtmp file. If the active*MMDD* file exists, check it first for error messages. If the active file and lock files exist, check fd2log for any mysterious messages. The following are error messages produced by *runacct*, and the recommended recovery actions:

ERROR: locks found, run aborted
The files lock and lock1 were found. These files must be removed before *runacct* can restart.

ERROR: acctg already run for *date* : check /usr/adm/acct/nite/lastdate
The date in lastdate and today's date are the same. Remove lastdate.

ERROR: turnacct switch returned rc = ?
Check the integrity of *turnacct* and *accton*. The *accton* program must be owned by root, and have the setuid bit set.

ERROR: Spacct*?*.*MMDD* already exists
file setups probably already run
Check status of files, then run setups manually.

ERROR: /usr/adm/acct/nite/wtmp.*MMDD* already exists, run setup manually

---

5. Accounting records are in *acct.h* format.

Self-explanatory.

ERROR: wtmpfix errors see /usr/adm/acct/nite/wtmperror
Wtmpfix detected a corrupted wtmp file. Use *fwtmp* to correct the corrupted file.

ERROR: connect acctg failed: check /usr/adm/acct/nite/log
The *acctcon1* program encountered a bad wtmp file. Use *fwtmp* to correct the bad file.

ERROR: Invalid state, check /usr/adm/acct/nite/active
The file, statefile, is probably corrupted. Check statefile and read active before restarting.

7. Restarting runacct

Runacct called without arguments assumes that this is the first invocation of the day. The argument *MMDD* is necessary if *runacct* is being restarted, and specifies the month and day for which *runacct* will rerun the accounting. The entry point for processing is based on the contents of statefile. To override statefile, include the desired *state* on the command line. Examples:

To start *runacct*
```
nohup runacct 2> /usr/adm/acct/nite/fd2log&
```

To restart *runacct*
```
nohup runacct 0601 2> /usr/adm/acct/nite/fd2log&
```

To restart *runacct* at a specific state:
```
nohup runacct 0601 WTMPFIX 2> /usr/adm/acct/nite/fd2log&
```

8. Fixing corrupted files

Unfortunately, this accounting system is not entirely fool proof. Occasionally a file will become corrupted, or lost. Some of the files can simply be ignored or restored from the filesave backup. However, certain files must be fixed in order to maintain the integrity of the accounting system.

The wtmp files seem to cause the most problems in the day to day operation of the accounting system. When the date is changed when UNIX is in multi-user mode, a set of date change records is written into /usr/adm/wtmp. The *wtmpfix* program is designed to adjust the time stamps in the wtmp records when a date change is encountered. Some combinations of date changes and reboots, however, will slip through *wtmpfix* and cause *acctcon1* to fail. The following steps show how to patch up a wtmp file.

```
ed /usr/adm/acct/nite
fwtmp < wtmp.MMDD > xwtmp
ed xwtmp
    delete corrupted records or
    delete all records from the beginning up to the date change
fwtmp -ic < xwtmp > wtmp.MMDD
```

If the wtmp file is beyond repair, create a null wtmp file. This will prevent any charging of connect time. *Acctprc1* won't be able to determine which login owned a particular process, but it will be charged to the login that is first in the password file for that userid.

If the installation is using the accounting system to charge users for system resources, the integrity of sum/tacct is quite important. Occasionally, mysterious tacct records will appear

with negative numbers, duplicate userids, or a userid of 65535. First check sum/tacctprev with *prtacct*. If it looks ok, the latest sum/tacct.*MMDD* should be patched up, then sum/tacct recreated. A simple patchup procedure would be:

```
cd /usr/adm/acct/sum
acctmerg -v < tacct.MMDD > xtacct
ed xtacct
   remove the bad records
   write duplicate uid records to another file
acctmerg -i < xtacct > tacct.MMDD
acctmerg tacctprev < tacct.MMDD > tacct
```

Remember that the *monacct* procedure removes all the tacct.*MMDD* files; therefore, sum/tacct can be recreated by merging these files together.

9. Recompiling pupsplit

The *pnpsplit* subroutine is used by *acctcon1* and *acctprc1* to determine the difference between prime and non-prime time. Prime time is defined as 9 a.m. to 5 p.m. Monday through Friday, holidays excluded. Every year on the day after Christmas, the following message will appear in log:

```
*** RECOMPILE pupsplit WITH NEW HOLIDAYS ***
```

Unfortunately, this will cause *runacct* to fail until *pnpsplit*, *acctcon1*, and *acctprc1* are recompiled. The following steps should be taken to successfully recompile these programs.

1. Edit pupsplit.c to change the thisyear variable to the new year. Update the holidays array to reflect the new holidays. Pupsplit.c is in /usr/src/cmd/acct/lib.
2. Recompile the accounting library *a.a*, *acctprc1*, and *acctcon1* by:

```
super-user to root
cd /usr/src
ARGS="library acctcon1 acctprc1" ./:makecmd acct
```

10. Summary

The PWB/UNIX accounting system was designed from a UNIX system administrator's point of view. Every possible precaution has been taken to ensure that the system will run smoothly and without error. It is important to become familiar with the C programs and shell procedures. The manual pages should be studied, and it is advisable to keep a printed copy of the shell procedures handy. This accounting system should be easy to maintain, provide valuable information for the administrator, and provide accurate breakdowns of the usage of system resources for charging purposes.

Files in the /usr/adm directory

diskdiag	diagnostic output during the execution of disk accounting programs
dtmp	output from the *acctdusg* program
fee	output from the *chargefee* program, ASCII tacct records
pacct	active process accounting file
pacct*?*	process accounting files switched via *turnacct*
Spacct*?.MMDD*	process accounting files for *MMDD* during execution of *runacct*
wtmp	active wtmp file for recording connect sessions

Files in the /usr/adm/acct/nite directory

active	used by *runacct* to record progress and print warning and error messages. **active***MMDD* same as active after *runacct* detects an error
cms	ASCII total command summary used by *prdaily*
ctacct.*MMDD*	connect accounting records in *tacct.h* format
ctmp	output of *acctcon1* program, connect session records in *ctmp.h* format
daycms	ASCII daily command summary used by *prdaily*
daytacct	total accounting records for one day in *tacct.h* format
disktacct	disk accounting records in *tacct.h* format, created by *dodisk* procedure
fd2log	diagnostic output during execution of runacct (see cron entry)
lastdate	last day *runacct* executed in date +%m%d format
lock lock1	used to control serial use of *runacct*
lineuse	tty line usage report used by *prdaily*
log	diagnostic output from *acctcon1*
log*MMDD*	same as log after *runacct* detects an error
reboots	contains beginning and ending dates from wtmp, and a listing of reboots
statefile	used to record current state during execution of *runacct*
tmpwtmp	wtmp file corrected by *wtmpfix*
wtmperror	place for *wtmpfix* error messages
wtmperror*MMDD*	same as wtmperror after *runacct* detects an error
wtmp.*MMDD*	previous day's wtmp file

Files in the /usr/adm/acct/sum directory

cms	total command summary file for current fiscal in internal summary format
cmsprev	command summary file without latest update
daycms	command summary file for yesterday in internal summary format
loginlog	created by *lastlogin*

pacct..*MMDD*	concatenated version of all pacct files for *MMDD*, removed after reboot by *remove* procedure
rprt..*MMDD*	saved output of *prdaily* program
tacct	cumulative total accounting file for current fiscal
tacctprev	same as tacct without latest update
tacct..*MMDD*	total accounting file for *MMDD*
wtmp..*MMDD*	saved copy of wtmp file for *MMDD*, removed after reboot by *remove* procedure

Files in the /usr/adm/acct/fiscal directory

cms*?*	total command summary file for fiscal *?* in internal summary format
fiscrpt*?*	report similar to *prdaily* for fiscal *?*
tacct*?*	total accounting file for fiscal *?*

Format of wtmp files (utmp.h)

```
/*
 * Format of /etc/utmp and /usr/adm/wtmp
 */

struct utmp {
        char    ut_line[8];             /* tty name */
        char    ut_name[8];             /* user id */
        long    ut_time;                /* time on */
};
```

Definitions (acctdef.h)

```
/*
 *      defines, typedefs, etc. used by acct programs
 */

/*
 *      following taken from (or modified versions of) <sys/types.h>
 */
typedef unsigned short  dev_t;
typedef unsigned int    ino_t;
typedef long            off_t;
typedef long            time_t;

/*
 *      acct only typedefs
 */
typedef unsigned short  uid_t;

#define LSZ     8       /* sizeof line name */
#define NSZ     8       /* sizeof login name */
#define P       0       /* prime time */
#define NP      1       /* nonprime time */

/*
 *      limits which may have to be increased if systems get larger
 */
#define SSIZE  1000     /* max number of sessions in 1 acct run */
#define TSIZE  100      /* max number of line names in 1 acct run */
#define USIZE 500       /* max number of distinct login names in 1 acct run */

#define EQN(s1, s2)     (strncmp(s1, s2, sizeof(s1)) == 0)
#define CPYN(s1, s2)    strncpy(s1, s2, sizeof(s1))

#define SECS(tics)      ((double) tics)/60.
#define MINS(secs)      ((double) secs)/60.
#define MINT(tics)      ((double) tics)/3600.
#define KCORE(clicks)           ((double) clicks/16)
#define SECSINDAY  86400L
```

Format of pacct files (acct.h)

```
/*
 * Accounting structures
 */

typedef unsigned short comp_t; /* "floating point" */

struct  acct
{
        char    ac_flag;        /* Accounting flag */
        char    ac_stat;        /* Exit status */
        short   ac_uid;                 /* Accounting user ID */
        short   ac_gid;                 /* Accounting group ID */
        dev_t   ac_tty;                 /* control typewriter */
        time_t  ac_btime;               /* Beginning time */
        comp_t  ac_utime;               /* Accounting user time */
        comp_t  ac_stime;               /* Accounting system time */
        comp_t  ac_etime;               /* Accounting elapsed time */
        comp_t  ac_mem;                         /* memory usage */
        comp_t  ac_io;
        comp_t  ac_rw;
        char    ac_comm[8];             /* command name */
};

extern  struct  acct    acctbuf;
extern  struct  inode   *acctp;         /* inode of accounting file */

#define AFORK           01              /* has executed fork, but no exec */
#define ASU     02              /* used super-user privileges */
#define ACCTF0300               /* record type: 00 = acct */
```

Format of tacct files (tacct.h)

```
/*
 *      total accounting (for acct period), also for day
 */

struct  tacct   {
        uid_t           ta_uid;         /* userid */
        char            ta_name[8];     /* login name */
        float           ta_cpu[2];      /* cum. cpu time, p/np (mins) */
        float           ta_kcore[2];    /* cum kcore-minutes, p/np */
        float           ta_con[2];      /* cum. connect time, p/np, mins */
        float           ta_du;          /* cum. disk usage */
        long            ta_pc;          /* count of processes */
        unsigned short  ta_sc;          /* count of login sessions */
        unsigned short  ta_dc;          /* count of disk samples */
        unsigned short  ta_fee;         /* fee for special services */
};
```

Format of cmap file (cmap.h)

```
/*
 *      connect time record (various intermediate files)
 */
struct ctmp {
        dev_t   ct_tty;                 /* major minor */
        uid_t   ct_uid;                 /* userid */
        char    ct_name[8];             /* login name */
        long    ct_con[2];              /* connect time (p/np) secs */
        time_t  ct_start;       /* session start time */
};
```

Jun 8 04:14 1979 DAILY REPORT FOR [illegible] Page 1

from Thu Jun 7 04:00:08 1979
to Fri Jun 8 04:00:28 1979
2 [illegible]
2 [illegible]

TOTAL DURATION IS 1320 MINUTES

LINE	MINUTES	PERCENT	# SESS	# ON	# OFF
cty04	479	36	9	9	30
cty47	341	26	4	4	33
cty44	298	23	3	3	29
cty46	336	25	9	9	33
[illegible]	1180	89	14	14	21
cty05	468	34	3	3	22
cty06	429	33	9	9	21
cty07	421	32	6	6	24
cty42	53	4	5	5	28
cty09	385	29	11	11	33
cty10	336	25	10	10	31
cty08	464	35	2	2	19
cty26	544	41	6	6	24
cty12	252	19	5	5	25
cty13	258	20	3	3	21
cty14	156	12	6	6	26
cty17	145	11	1	1	16
cty18	39	3	5	5	24
cty15	228	17	5	5	25
cty25	704	53	6	6	25
cty21	0	0	0	0	16
cty19	10	1	1	1	17
cty20	25	2	2	2	18
cty22	0	0	0	0	15
cty23	0	0	0	0	15
cty24	0	0	0	0	16
cty27	481	36	3	3	20
cty28	426	32	5	5	24
cty29	302	23	6	6	25
cty30	257	20	11	11	28
cty40	380	29	5	5	21
cty41	343	26	3	3	21
cty43	0	0	0	0	15
cty11	365	28	7	7	25
cty45	3	0	1	1	17
cty16	213	16	3	3	20
cty31	250	19	4	4	18
cty02	62	5	1	1	3
TOTALS	10544	-	174	174	844

Jan 8 04:14 1979 DAILY USAGE REPORT FOR [illegible] Page 1

UID	LOGIN NAME	CPU (MINS) PRIME	CPU (MINS) NPRIME	KCORE-MINS PRIME	KCORE-MINS NPRIME	CONNECT (MINS) PRIME	CONNECT (MINS) NPRIME	DISK BLOCKS	# OF PROCS	# OF SESS	# DISK SAMPLES	FEE
0	TOTAL	388	103	12414	2934	9251	1056	0	16164	174	0	0
0	[illegible]	47	41	10803	924	67	38	0	2348	8	0	0
4	[illegible]	7	19	48	652	0	0	0	842	0	0	0
19	[illegible]	0	0	4	0	0	0	0	28	0	0	0
21	[illegible]	0	0	1	1	1	1	0	14	2	0	0
17	[illegible]	0	0	4	0	0	0	0	3	0	0	0
37	[illegible]	14	0	284	0	422	0	0	1588	4	0	0
68	[illegible]	1	3	26	11	0	3	0	179	0	0	0
71	?	0	0	0	0	0	0	0	12	0	0	0
150	[illegible]	7	0	155	5	231	2	0	510	13	0	0
173	?	0	0	0	0	0	0	0	16	0	0	0
159	?	0	0	0	0	0	0	0	4	0	0	0
185	?	0	0	0	0	0	0	0	2	0	0	0
217	[illegible]	0	0	2	0	31	0	0	32	3	0	0
217	[illegible]	0	0	2	0	1	0	0	7	1	0	0
219	?	0	0	0	0	0	0	0	12	0	0	0
1031	[illegible]	5	0	179	0	179	0	0	93	2	0	0
2031	[illegible]	0	1	5	28	476	64	0	99	5	0	0
2032	[illegible]	1	0	7	3	270	42	0	93	3	0	0
2033	[illegible]	1	0	23	0	108	0	0	99	3	0	0
2035	[illegible]	0	0	3	0	13	0	0	21	1	0	0
2036	[illegible]	0	0	2	0	14	0	0	8	1	0	0
2039	[illegible]	47	0	2048	0	444	0	0	492	2	0	0
2039	[illegible]	2	0	68	0	34	0	0	95	1	0	0
2041	[illegible]	0	0	1	0	4	0	0	11	1	0	0
2042	[illegible]	0	0	1	0	28	0	0	9	1	0	0
2044	[illegible]	0	0	1	0	1	0	0	7	1	0	0
2052	[illegible]	32	1	1227	31	578	4	0	226	3	0	0
2025	[illegible]	55	23	2178	882	336	98	0	758	7	0	0
2027	[illegible]	14	2	368	51	547	24	0	372	8	0	0
2028	[illegible]	0	0	1	0	1	0	0	13	1	0	0
2030	[illegible]	8	0	288	0	317	0	0	315	3	0	0
2031	[illegible]	12	0	468	3	459	6	0	228	6	0	0
2032	[illegible]	2	0	48	0	23	0	0	118	1	0	0
2033	[illegible]	1	0	76	0	351	0	0	113	4	0	0
2040	[illegible]	15	0	368	0	313	1	0	305	2	0	0
2041	[illegible]	1	0	93	3	146	2	0	117	2	0	0
2051	[illegible]	2	2	19	48	379	681	0	611	5	0	0
2055	[illegible]	0	0	15	9	17	1	0	10	3	0	0
2057	[illegible]	1	0	23	0	63	0	0	68	2	0	0
2061	[illegible]	4	3	99	70	37	34	0	159	4	0	0
2064	[illegible]	18	0	648	0	178	0	0	2865	3	0	0
2065	[illegible]	0	0	6	0	7	0	0	23	1	0	0
2068	[illegible]	0	0	7	0	10	0	0	29	1	0	0
2073	[illegible]	28	0	1178	1	384	2	0	249	5	0	0
2024	[illegible]	1	0	14	0	152	0	0	28	1	0	0
2087	[illegible]	0	0	0	10	0	2	0	13	1	0	0
2111	[illegible]	3	3	59	85	64	86	0	185	4	0	0
2114	[illegible]	1	0	16	0	488	0	0	222	1	0	0
2121	[illegible]	0	0	3	0	33	0	0	58	2	0	0
2123	[illegible]	0	0	3	0	5	0	0	24	1	0	0
2124	[illegible]	2	0	42	0	66	0	0	160	3	0	0
2126	[illegible]	0	0	3	0	121	0	0	17	1	0	0
2127	[illegible]	13	0	495	11	398	2	0	882	10	0	0

Jan 8 04:14 1979 DAILY USAGE REPORT FOR [illegible] Page 2

2128	[illegible]	14	0	492	9	422	14	0	523	8	0	0
2130	[illegible]	0	0	5	1	16	0	0	42	2	0	0
2130	[illegible]	0	0	0	0	0	2	0	9	1	0	0
2135	[illegible]	0	1	0	12	0	11	0	33	2	0	0
2136	[illegible]	0	0	5	0	2	0	0	18	1	0	0
2140	[illegible]	4	0	213	12	98	3	0	170	7	0	0
2141	[illegible]	5	0	248	25	478	4	0	181	1	0	0
2159	[illegible]	0	0	1	0	10	0	0	7	1	0	0
2249	[illegible]	0	0	1	0	1	0	0	17	1	0	0
3001	[illegible]	5	0	98	0	252	0	0	414	3	0	0
1222	[illegible]	0	0	4	0	8	0	0	39	1	0	0

Jun 8 04:07 1979 DAILY CORMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCORE-MIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	16164	15332.89	490.72	37453.98	31.25	0.03	0.01	3221833564	1097670
[illegible]	119	3958.68	98.21	569.83	42.47	0.78	0.16	670702052	130284
[illegible]	26	2483.38	51.63	343.70	48.10	1.99	0.15	378369304	48949
[illegible]	20	732.03	16.74	111.05	43.73	0.84	0.15	138815348	22659
[illegible]	31	623.33	18.52	142.77	33.65	0.34	0.57	3524435	2758
[illegible]	185	574.83	13.96	34.53	41.18	0.08	0.40	1706625	8269
[illegible]	232	555.79	9.93	155.11	55.96	0.04	0.06	6155937	20594
[illegible]	150	519.84	13.57	48.89	38.25	0.09	0.28	4285724	16032
[illegible]	165	413.10	9.19	35.16	44.93	0.05	0.26	3827309	13170
[illegible]	33	340.93	6.63	148.77	73.62	0.14	0.03	1074914	14492
[illegible]	87	317.38	7.94	28.48	39.97	0.09	0.21	17640396	45797
[illegible]	17	294.75	6.49	14.15	45.41	0.38	0.46	2525427	5515
[illegible]	112	289.69	9.13	34.61	31.72	0.08	0.26	3557050	9681
[illegible]	1834	276.91	26.77	28044.24	10.35	0.01	0.00	3496613	71979
[illegible]	324	253.13	14.46	2523.89	17.50	0.03	0.01	150581108	56039
[illegible]	3	231.28	6.67	19.45	34.67	2.22	0.34	2577344	2926
[illegible]	145	219.35	19.91	59.08	11.02	0.14	0.51	716389	23695
[illegible]	49	175.53	6.04	25.78	29.05	0.12	0.23	3740587	11351
[illegible]	151	152.96	4.28	25.23	35.74	0.03	0.17	3534042	24917
[illegible]	23	148.10	4.07	202.35	36.37	0.19	0.02	2313718	9813
[illegible]	24	143.43	2.44	210.65	58.71	0.10	0.01	1536210	3433
[illegible]	9	139.24	10.15	51.05	13.72	1.13	0.20	24006848	394
[illegible]	155	129.33	9.82	42.25	13.17	0.06	0.23	10500835	30165
[illegible]	397	115.46	4.19	36.23	27.57	0.01	0.12	783825	26497
[illegible]	51	109.49	5.92	41.35	18.54	0.12	0.14	2278056	8310
[illegible]	89	102.94	2.87	202.32	35.81	0.03	0.01	1018461	2564
[illegible]	25	90.23	2.27	17.80	39.70	0.09	0.13	2907269	9321
[illegible]	172	89.37	2.69	11.32	33.19	0.02	0.24	3519054	12153
[illegible]	16	86.94	1.30	10.57	66.85	0.08	0.12	27671849	2927
[illegible]	52	85.64	5.05	63.87	17.15	0.10	0.08	563125	11161
[illegible]	706	82.47	5.78	62.85	14.26	0.01	0.09	1811882	29659
[illegible]	2	79.44	10.49	47.89	7.57	5.25	0.22	195016	21995
[illegible]	22	78.83	1.37	5.26	57.72	0.06	0.26	355466	3769
[illegible]	60	75.55	1.42	672.50	53.27	0.02	0.00	390693	4377
[illegible]	9	75.21	2.81	11.49	26.75	0.31	0.24	1283776	3771
[illegible]	2814	66.10	7.08	91.80	9.33	0.00	0.08	166651	24253
[illegible]	3	57.27	0.82	7.51	70.16	0.27	0.11	51832	426
[illegible]	284	56.92	2.42	9.43	23.48	0.01	0.26	15283	20329
[illegible]	7	48.03	6.80	84.45	7.06	0.97	0.08	279451	1700
[illegible]	749	45.49	5.69	471.54	8.00	0.01	0.01	8959500	27903
[illegible]	6	41.52	1.55	7.55	26.87	0.26	0.20	59288	478
[illegible]	202	39.95	2.05	232.98	19.53	0.01	0.00	427217	14377
[illegible]	3	38.95	1.43	19.45	27.24	0.48	0.07	587336	87
[illegible]	94	38.72	1.09	9.73	35.41	0.01	0.11	375876	4423
[illegible]	104	34.89	2.47	214.50	14.10	0.02	0.01	1040989	6572
[illegible]	7	33.20	5.28	1244.54	6.29	0.75	0.00	63064	35633
[illegible]	17	31.69	0.62	41.84	50.97	0.04	0.02	514624	3393
[illegible]	213	28.73	2.98	21.01	9.64	0.01	0.14	2109229	14297

Jan 8 0:57 1979 MONTHLY TOTAL COMMAND SUMMARY Page 1

COMMAND NAME	NUMBER CMDS	TOTAL KCORE-MIN	TOTAL CPU-MIN	TOTAL REAL-MIN	MEAN SIZE-K	MEAN CPU-MIN	HOG FACTOR	CHARS TRNSFD	BLOCKS READ
TOTALS	557256	397698.78	10916.89	745924.94	37.27	0.02	0.01	820472548	26253312
[illegible]	1637	44681.35	595.92	5737.25	44.86	0.39	0.17	613403153	1089160
[illegible]	1351	25693.15	583.69	4156.05	44.02	0.43	0.13	413161589	645243
[illegible]	6466	17798.41	294.16	1893.79	58.81	0.05	0.16	334572640	853901
[illegible]	654	13526.69	169.62	4238.58	82.17	0.25	0.04	54940628	427924
[illegible]	397	10408.44	228.72	1496.12	51.09	0.51	0.14	215231419	301947
[illegible]	7588	9892.36	256.81	2298.05	41.11	0.03	0.10	80108304	353943
[illegible]	6139	9369.86	256.48	661.09	37.85	0.04	0.27	79819995	459661
[illegible]	3266	8852.96	233.19	1128.09	39.67	0.07	0.20	493701995	1271119
[illegible]	53134	8126.71	313.35	2241.78	25.89	0.01	0.14	23035033	1697990
[illegible]	2582	7934.45	149.18	1694.25	53.16	0.05	0.08	1113300040	444504
[illegible]	6126	7864.42	185.16	725.47	42.49	0.03	0.26	72595635	219426
[illegible]	20583	7822.78	425.53	41898.18	18.37	0.02	0.01	483425634	1341326
[illegible]	569	7765.69	113.93	2658.53	68.16	0.17	0.05	50760094	13187
[illegible]	45876	7469.67	635.03	1217826.53	11.81	0.02	0.09	70925236	1421194
[illegible]	1941	6730.56	153.04	1128.44	12.17	0.28	0.49	20848359	618324
[illegible]	1403	5638.46	125.87	1868.87	44.69	0.09	0.07	16152675	87260
[illegible]	4291	5573.31	139.89	460.25	39.85	0.03	0.30	6823696	237298
[illegible]	793	5325.66	71.23	425.67	74.76	0.09	0.17	9599001	131593
[illegible]	21170	4857.53	236.59	4254.24	19.69	0.01	0.05	239180412	1023961
[illegible]	42	3837.84	110.88	291.34	34.61	2.64	0.38	43954136	61123
[illegible]	4087	3697.23	144.26	477.28	25.28	0.04	0.30	575319276	213521
[illegible]	21212	3206.86	369.66	2727.87	18.67	0.01	0.11	139340583	899415
[illegible]	7469	3064.72	94.12	647.79	32.52	0.01	0.15	91471956	459482
[illegible]	15556	2948.71	853.53	101107.45	3.45	0.02	0.01	34704751	263866
[illegible]	53	2707.27	28.79	181.84	94.02	0.55	0.08	2852202	33949
[illegible]	6454	2698.74	218.35	910.59	12.33	0.03	0.24	213336016	705690
[illegible]	1258	2669.18	64.69	4588.86	37.26	0.05	0.01	24116259	175544
[illegible]	1034	2184.14	128.29	1207.87	18.58	0.12	0.11	54873791	254172
[illegible]	294	2288.16	51.99	116.06	44.01	0.18	0.45	36124940	90523
[illegible]	815	2256.79	45.42	11729.01	53.08	0.05	0.00	11086105	162558
[illegible]	14276	2170.01	152.76	1528.09	14.20	0.01	0.10	32418106	691028
[illegible]	1705	2114.18	114.35	920.75	18.49	0.07	0.12	94431159	338600
[illegible]	72	2028.43	28.54	317.21	71.01	0.40	0.09	1644636	10374
[illegible]	56710	2018.23	190.14	1138.49	10.61	0.00	0.17	2926992	649200
[illegible]	127	1956.68	77.03	391.45	25.40	0.61	0.20	190822344	236302
[illegible]	8	1620.42	44.08	128.25	36.17	5.50	0.35	120399	112
[illegible]	4735	1474.38	76.92	14261.82	19.17	0.02	0.01	25719618	463748
[illegible]	1695	1353.08	37.59	234.97	36.13	0.02	0.16	31540008	178613
[illegible]	165	1253.59	47.05	339.34	26.64	0.29	0.14	57405662	68549
[illegible]	58	1187.17	15.36	36.90	77.31	0.26	0.42	4096070	12093
[illegible]	638	1064.43	49.81	2199.09	21.72	0.08	0.02	23833295	16933
[illegible]	27136	1036.03	93.14	1941.33	11.12	0.00	0.05	925447	259143
[illegible]	29	909.83	17.71	34.97	51.38	0.61	0.51	11459928	15802
[illegible]	264	904.56	23.07	254.06	39.21	0.09	0.09	24219141	57164
[illegible]	173	868.19	25.74	159.73	34.43	0.15	0.16	1990177	15792
[illegible]	1436	872.65	61.87	309.07	14.11	0.04	0.20	189838731	428871
[illegible]	144	864.29	12.54	344.13	68.94	0.09	0.04	1186547	28576
[illegible]	15319	857.97	85.65	754.20	10.02	0.01	0.11	453479	433903
[illegible]	1	819.77	19.30	170.10	39.86	19.30	0.23	1812480	19744
[illegible]	155	779.13	7.97	29.09	97.70	0.05	0.27	960027	34702
[illegible]	756	767.31	33.77	269.27	23.41	0.04	0.13	22940954	97214

Jun 8 04:07 1979 LAST LOGIN Page 1

00-00-00	dii	00-00-00	rodd	79-06-08	adm
00-00-00	abradm	00-00-00	s10	79-06-08	agp
00-00-00	abssf?	00-00-00	s2	79-06-08	ale
00-00-00	absous	00-00-00	s4	79-06-08	arf
00-00-00	abnew	00-00-00	s5	79-06-08	ash
00-00-00	abnpwg	00-00-00	s6	79-06-08	dal
00-00-00	abstbm	00-00-00	s8	79-06-08	dcn
00-00-00	adm94	00-00-00	s9	79-06-08	donnm
00-00-00	apb	00-00-00	sdsm	79-06-08	dp
00-00-00	archive	00-00-00	sjm	79-06-08	ds52
00-00-00	sac	00-00-00	srb	79-06-08	esh
00-00-00	badi	00-00-00	syn	79-06-08	ecp
00-00-00	bib	00-00-00	tgp	79-06-08	eric
00-00-00	bwl	00-00-00	tld	79-06-08	fbd
00-00-00	bwk	00-00-00	tmc	79-06-08	ferep1
00-00-00	chicken	00-00-00	tsmps	79-06-08	games
00-00-00	class	00-00-00	uvac	79-06-08	graf
00-00-00	cleary	00-00-00	vev	79-06-08	harb
00-00-00	cn	00-00-00	vdr	79-06-08	hcot
00-00-00	dbs	00-00-00	willa	79-06-08	hzm
00-00-00	coby	00-00-00	zscnma	79-06-08	jac
00-00-00	dsc	79-06-04	dws	79-06-08	jcw
00-00-00	demo	79-06-04	evb	79-06-08	jel
00-00-00	dlt	79-06-04	jms	79-06-08	jfa
00-00-00	dmr	79-06-04	smz	79-06-08	kaf
00-00-00	docs	79-06-04	wmp	79-06-08	krb
00-00-00	dug	79-06-05	bcm	79-06-08	lmp
00-00-00	ellie	79-06-05	tprom	79-06-08	llc
00-00-00	ferep2	79-06-05	s7	79-06-08	m2
00-00-00	gas	79-06-05	zeez	79-06-08	m2class
00-00-00	graphics	79-06-06	ssrv	79-06-08	mfp
00-00-00	hig	79-06-06	dek	79-06-08	mbb
00-00-00	hlb	79-06-06	dmi	79-06-08	mjr
00-00-00	isat	79-06-06	emp	79-06-08	mab
00-00-00	jfm	79-06-06	peh	79-06-08	muucp
00-00-00	jrb	79-06-06	sync	79-06-08	paul
00-00-00	ken	79-06-06	tad	79-06-08	pdw
00-00-00	leo	79-06-C'	ams	79-06-08	prs
00-00-00	mara	79-06-07	hin	79-06-08	pwbcs
00-00-00	mpdw	79-06-07	dgd	79-06-08	pwbus
00-00-00	trbo	79-06-07	bright	79-06-08	rbj
00-00-00	mau	79-06-07	hzsp	79-06-08	rag
00-00-00	mar	79-06-07	jgw	79-06-08	rech
00-00-00	mash	79-06-07	lcb	79-06-08	rjs
00-00-00	meq	79-06-07	ljk	79-06-08	rst
00-00-00	mib	79-06-07	mcp	79-06-08	root
00-00-00	mlc	79-06-07	nbg	79-06-08	rrr
00-00-00	mmr	79-06-07	gws	79-06-08	s1
00-00-00	mpf	79-06-07	gnroff	79-06-08	s3
00-00-00	plan	79-06-07	thm	79-06-08	sar
00-00-00	plum	79-06-07	train	79-06-08	stock
00-00-00	pvg	79-06-07	whr	79-06-08	sysust
00-00-00	rebcab	79-06-07	wcw	79-06-08	teach
00-00-00	rfg	79-06-08	?	79-06-08	text
00-00-00	ric	79-06-08	abs	79-06-08	trc
00-00-00	rrc	79-06-08	abspk	79-06-08	vjf
79-06-08	wbm				

FSCK—The UNIX/TS File System Check Program

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

When a UNIX/TS operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. No changes are made to any file system by *fsck* without prior operator approval.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of heuristically sound corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. UPDATE OF THE FILE SYSTEM

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the super-block, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

2.1 Super-Block

The super-block contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The super-block of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a *sync* command is issued.

2.2 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system upon closure[1] of the file associated with the inode.

2.3 Indirect Blocks

There are three types of indirect blocks: single-indirect, double-indirect and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each one of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released by the operating system.

2.4 Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.

2.5 First Free-List Block

The super-block contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the super-block, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system whenever they have been modified and released by the operating system.

3. CORRUPTION OF THE FILE SYSTEM

A file system can become corrupted in a variety of ways. The most common of these ways are improper shutdown procedures and hardware failures.

3.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to *sync* the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

3.2 Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

4. DETECTION AND CORRECTION OF CORRUPTION

A quiescent[3] file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multi-pass nature of the *fsck* program.

When an inconsistency is discovered *fsck* reports the inconsistency for the operator to chose a corrective action.

Discussed in this section are how to discover inconsistencies and possible corrective actions for the super-block, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the *fsck* command under control of the operator.

---

1. All in core blocks are also written to the file system upon issue of a *sync* system call.
2. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by UNIX/TS or a *sync* command is issued.
3. I.e., unmounted and not being written on.

4.1 Super-Block

One of the most common corrupted items is the super-block. The super-block is prone to corruption because every change to the file system's blocks or inodes modifies the super-block.

The super-block and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a *sync* command.

The super-block can be checked for inconsistencies involving file-system size, inode-list size, free-block list, free-block count, and the free-inode count.

4.1.1 File-System Size and Inode-List Size. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535. The file-system size and inode-list size are critical pieces of information to the *fsck* program. While there is no way to actually check these sizes, *fsck* can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

4.1.2 Free-Block List. The free-block list starts in the super-block and continues through the free-list blocks of the file system. Each free-list block can be checked for a list count out of range, for block numbers out of range, and for blocks already allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free-block list is in the super-block. *Fsck* checks the list count for a value of less than zero or greater than fifty. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is non-zero, the next free-list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the free-block list, then *fsck* may rebuild it, excluding all blocks in the list of allocated blocks.

4.1.3 Free-Block Count. The super-block contains a count of the total number of free blocks within the file system. *Fsck* compares this count to the number of blocks it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-block count.

4.1.4 Free-Inode Count. The super-block contains a count of the total number of free inodes within the file system. *Fsck* compares this count to the number of inodes it found free within the file system. If they don't agree, then *fsck* may replace the count in the super-block by the actual free-inode count.

4.2 Inodes

An individual inode is not as likely to be corrupted as the super-block. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the super-block.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

4.2.1 Format and Type. Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes may be one of four types: regular inode, directory inode, special block inode, and special character inode. If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states: unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An

inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for *fsck* is to clear the inode.

4.2.2 Link Count. Contained in each inode is a count of the total number of directory entries linked to the inode.

Fsck verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, calculating an actual link count for each inode.

If the stored link count is non-zero and the actual link count is zero, it means that no directory entry appears for the inode. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is non-zero and the actual link count is zero, *fsck* may link the disconnected file to the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, *fsck may* replace the stored link count by the actual link count.

4.2.3 Duplicate Blocks. Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode.

Fsck compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number. If there are any duplicate blocks, *fsck* will make a partial second pass of the inode list to find the inode of the duplicated block, because without examining the files associated with these inodes for correct content, there is not enough information available to decide which inode is corrupted and should be cleared. Most times, the inode with the earliest modify time is incorrect, and should be cleared.

This condition can occur by using a file system with blocks claimed by both the free-block list and by other parts of the file system.

If there is a large number of duplicate blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.4 Bad Blocks. Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode.

Fsck checks each block number claimed by an inode for a value lower than that of the first data block, or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system.

Fsck will prompt the operator to clear both inodes.

4.2.5 Size Checks. Each inode contains a thirty-two bit (four-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of sixteen characters, or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the UNIX file system has the directory bit on in the inode mode word. The directory size must be a multiple of sixteen because a directory entry contains sixteen bytes of information (two bytes for the inode number and fourteen bytes for the file or directory name).

Fsck will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in a inode by the number of characters per block (512) and rounding up. *Fsck* adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, *fsck* will warn of a possible file-size error. This is only a warning because UNIX/TS does not fill in blocks in files created in random order.

4.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, apply iteratively Sections 4.2.3 and 4.2.4 to each level of indirect blocks.

4.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories which are disconnected from the file system.

If a directory entry inode number points to an unallocated inode, then *fsck* may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written to the file system while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, *fsck* may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory).

If the directory inode numbers are incorrect, *fsck* may replace them by the correct values.

Fsck checks the general connectivity of the file system. If directories are found not to be linked into the file system, *fsck* will link the directory back into the file system in the *lost+found* directory. This condition can be caused by inodes being written to the file system with the corresponding directory data blocks not being written to the file system.

4.5 Free-List Blocks

Free-list blocks are owned by the super-block. Therefore, inconsistencies in free-list blocks directly affect the super-block.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks see Section 4.1.2.

ACKNOWLEDGEMENT

I would like to thank Larry A. Wehr for advice that lead to the first version of *fsck* and Rick B. Brandt for adapting *fsck* to UNIX/TS.

REFERENCES

[1] Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.

[2] Deloria, T. A., and Olsson, S. B. eds., *UNIX/TS User's Manual, Edition 1.1* (January 1978).

[3] Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

Appendix—FSCK ERROR CONDITIONS

1. CONVENTIONS

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the free-block list (possibly rebuilding it), and performs some cleanup.

When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

2. INITIALIZATION

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file.

C option?

C is not a legal option to *fsck*; legal options are −y, −n, −s, −S, and −t. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

Bad −t option

The −t option is not followed by a file name. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

Invalid −s argument, defaults assumed

The −s option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. *Fsck* assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-skip. See the *fsck*(1M) manual entry for more details.

Incompatible options: −n and −s

It is not possible to salvage the free-block list without modifying the file system. *Fsck* terminates on this error condition. See the *fsck*(1M) manual entry for further detail.

Can't get memory

Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't open checklist file: F

The default file system checklist file F (usually */etc/checklist*) can not be opened for reading. *Fsck* terminates on this error condition. Check access modes of F.

Can't stat root

Fsck's request for statistics about the root directory "/" failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't stat F

Fsck's request for statistics about the file system F failed. It ignores this file system and continues checking the next file system given. Check access modes of F.

F is not a block or character device

You have given *fsck* a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of F.

Can't open F

The file system F can not be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of F.

Size check: fsize X isize Y

More blocks are used for the inode list Y than there are blocks in the file system X, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given. See Section 4.1.1.

Can't create F

Fsck's request to create a scratch file F failed. It ignores this file system and continues checking the next file system given. Check access modes of F.

CAN NOT SEEK: BLK B (CONTINUE)

Fsck's request for moving to a specified block number B in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT READ: BLK B (CONTINUE)

Fsck's request for reading a specified block number B in the file system failed. This should never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

CAN NOT WRITE: BLK B (CONTINUE)

Fsck's request for writing a specified block number B in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If the block was part of the virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO terminate the program.

3. PHASE 1: CHECK BLOCKS AND SIZES

This phase concerns itself with the inode list. This section lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

UNKNOWN FILE TYPE I=I (CLEAR)

The mode word of the inode I indicates that the inode is not a special character inode, special character inode, regular inode, or directory inode. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)

An internal table for *fsck* containing allocated inodes with a link count of zero has no more room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another allocated inode with a zero link count is found, this error condition is repeated.

NO terminate the program.

B BAD I=I

Inode I contains block number B with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the EXCESSIVE BAD BLKS error condition in Phase 1 if inode I has too many block numbers outside the file system range. This error condition will always invoke the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.4.

EXCESSIVE BAD BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode I. See Section 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

B DUP I=I

Inode I contains block number B which is already claimed by another inode. This error condition may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode I has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4. See Section 4.2.3.

EXCESSIVE DUP BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes. See Section 4.2.3.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO terminate the program.

DUP TABLE OVERFLOW (CONTINUE)

An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO terminate the program.

POSSIBLE FILE SIZE ERROR I=I

The inode I size does not match the actual number of blocks used by the inode. This is only a warning. See Section 4.2.5.

DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. See Section 4.2.5.

PARTIALLY ALLOCATED INODE I=I (CLEAR)

Inode I is neither allocated nor unallocated. See Section 4.2.1.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.
NO ignore this error condition.

4. PHASE 1B: RESCAN FOR MORE DUPS

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

B DUP I=I

Inode I contains block number B which is already claimed by another inode. This error condition will always invoke the BAD/DUP error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1. See Section 4.2.3.

5. PHASE 2: CHECK PATH-NAMES

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

ROOT INODE UNALLOCATED. TERMINATING.

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate. See Section 4.2.1.

ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type. See Section 4.2.1.

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.
NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system. See Section 4.2.3 and 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.
NO terminate the program.

I OUT OF RANGE I=I NAME=F (REMOVE)

A directory entry F has an inode number I which is greater than the end of the inode list. See Section 4.4.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.
NO ignore this error condition.

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)

A directory entry F has an inode I without allocate mode bits. The owner O, mode M, size S, modify time T, and file name F are printed. See Section 4.4.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.
NO ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry F, directory inode I. The owner O, mode M, size S, modify time T, and directory name F are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.
NO ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry F, inode I. The owner O, mode M, size S, modify time T, and file name F are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.
NO ignore this error condition.

6. PHASE 3: CHECK CONNECTIVITY

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)

The directory inode I was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of directory inode I are printed. See Section 4.4 and 4.2.2.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode I to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode I to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck*(1M) manual entry for further detail.

SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See *fsck*(1M) manual entry for further detail.

DIR I=I1 CONNECTED. PARENT WAS I=I2

This is an advisory message indicating a directory inode I1 was successfully connected to the *lost+found* directory. The parent inode I2 of the directory inode I1 is replaced by the inode number of the *lost+found* directory. See Section 4.4 and 4.2.2.

7. PHASE 4: CHECK REFERENCE COUNTS

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, or special files, unreferenced files and directories, bad and duplicate blocks in files and directories, and incorrect total free-inode counts.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)

Inode I was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2.

Possible responses to the RECONNECT prompt are:

YES reconnect inode I to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode I to *lost+found*.

NO ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

SORRY. NO lost+found DIRECTORY

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found*.

SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*.

(CLEAR)

The inode mentioned in the immediately previous error condition can not be reconnected. See Section 4.2.2.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

NO ignore this error condition.

LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for inode I which is a file, is X but should be Y. The owner O, mode M, size S, and modify time T are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode I with Y.

NO ignore this error condition.

LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for inode I which is a directory, is X but should be Y. The owner O, mode M, size S, and modify time T of directory inode I are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES	replace the link count of directory inode I with Y.
NO	ignore this error condition.

LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for F inode I is X but should be Y. The name F, owner O, mode M, size S, and modify time T are printed. See Section 4.2.2.

Possible responses to the ADJUST prompt are:

YES	replace the link count of inode I with Y.
NO	ignore this error condition.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Inode I which is a file, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES	de-allocate inode I by zeroing its contents.
NO	ignore this error condition.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Inode I which is a directory, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.2 and 4.4.

Possible responses to the CLEAR prompt are:

YES	de-allocate inode I by zeroing its contents.
NO	ignore this error condition.

BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode I. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES	de-allocate inode I by zeroing its contents.
NO	ignore this error condition.

BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode I. The owner O, mode M, size S, and modify time T of inode I are printed. See Section 4.2.3 and 4.2.4.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.
NO ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX)

The actual count of the free inodes does not match the count in the super-block of the file system. See Section 4.1.4.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.
NO ignore this error condition.

8. PHASE 5: CHECK FREE LIST

This phase concerns itself with the free-block list. This section lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and the total free-block count incorrect.

EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system. See Section 4.1.2 and 4.2.4.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the BAD BLKS IN FREE LIST error condition in Phase 5.
NO terminate the program.

EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list. See Section 4.1.2 and 4.2.3.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block list and continue the execution of *fsck*. This error condition will always invoke the DUP BLKS IN FREE LIST error condition in Phase 5.
NO terminate the program.

BAD FREEBLK COUNT

The count of free blocks in a free-list block is greater than 50 or less than zero. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

X BAD BLKS IN FREE LIST

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.4.

X DUP BLKS IN FREE LIST

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2 and 4.2.3.

X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the BAD FREE LIST condition in Phase 5. See Section 4.1.2.

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)

The actual count of free blocks does not match the count in the super-block of the file system. See Section 4.1.3.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.
NO ignore this error condition.

BAD FREE LIST (SALVAGE)

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system. See Section 4.1.2, 4.2.3, and 4.2.4.

Possible responses to the SALVAGE prompt are:

YES replace the actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.
NO ignore this error condition.

9. PHASE 6: SALVAGE FREE LIST

This phase concerns itself with the free-block list reconstruction. This section lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

Default free-block list spacing assumed

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than one, the blocks-per-cylinder is less than one, or the blocks-per-cylinder is greater than 500. The default values of 9 blocks-to-skip and 400 blocks-per-cylinder are used. See the *fsck*(1M) manual entry for further detail.

10. CLEANUP

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

X files Y blocks Z free

This is an advisory message indicating that the file system checked contained X files using Y blocks leaving Z blocks free in the file system.

******* BOOT UNIX (NO SYNC!) *******

This is an advisory message indicating that a mounted file system or the root file system has been modified by *fsck*. If UNIX/TS is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX/TS keeps.

******* FILE SYSTEM WAS MODIFIED *******

This is an advisory message indicating that the current file system was modified by *fsck*. If this file system is mounted or is the current root file system, *fsck* should be halted and UNIX/TS rebooted. If UNIX/TS is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX/TS keeps.

INDEX OF MESSAGES
(Alphabetically within each section)

INITIALIZATION

PHASE 1: CHECK BLOCKS AND SIZES

PHASE 1B: RESCAN FOR MORE DUPS

PHASE 2: CHECK PATH-NAMES

PHASE 3: CHECK CONNECTIVITY

PHASE 4: CHECK REFERENCE COUNTS

PHASE 5: CHECK FREE LIST

PHASE 6: SALVAGE FREE LIST

CLEANUP

The UNIX I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after

†UNIX is a Trademark of Bell Laboratories.

the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base*, *u.u_count*, and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a

flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on*, that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count*.

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p)(dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of

characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a

residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread*, but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread*. There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTEDThis flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRIThis bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back)*, and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat*, the buffer pointer *bp*, the device number *dev*, and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE*. *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous

†UNIX is a Trademark of Bell Laboratories.

execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory.. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

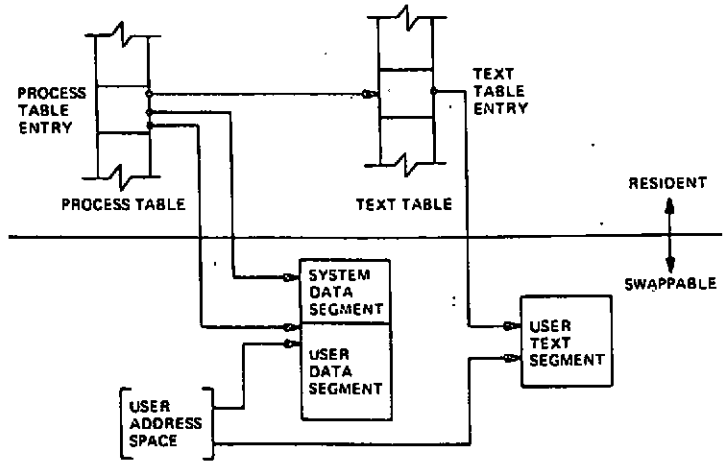


Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **exec**s the second program. This is analogous to a "goto." If a program wishes to regain control after execing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character

I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open, create, read, write, seek,** and **close.** The data structures maintained are shown in Fig. 2.

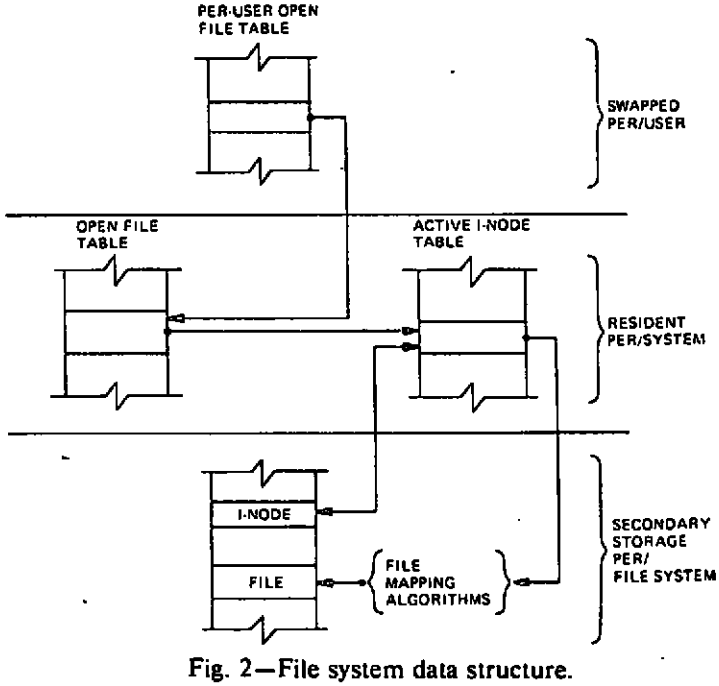


Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer

and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems.

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user

code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices* **12**(4) pp. 40-50 (April 1977).
2. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, New York (1968).
3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).
4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).
5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6) pp. 1971-1990 (1978).
6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).

A Tour Through the Portable C Compiler

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

A C compiler has been implemented that has proved to be quite portable, serving as the basis for C compilers on roughly a dozen machines, including the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler is highly compatible with the C language standard.[1]

Among the goals of this compiler are portability, high reliability, and the use of state-of-the-art techniques and tools wherever practical. Although the efficiency of the compiling process is not a primary goal, the compiler is efficient enough, and produces good enough code, to serve as a production compiler.

The language implemented is highly compatible with the current PDP-11 version of C. Moreover, roughly 75% of the compiler, including nearly all the syntactic and semantic routines, is machine independent. The compiler also serves as the major portion of the program *lint*, described elsewhere.[2]

A number of earlier attempts to make portable compilers are worth noting. While on CO-OP assignment to Bell Labs in 1973, Alan Snyder wrote a portable C compiler which was the basis of his Master's Thesis at M.I.T.[3] This compiler was very slow and complicated, and contained a number of rather serious implementation difficulties; nevertheless, a number of Snyder's ideas appear in this work.

Most earlier portable compilers, including Snyder's, have proceeded by defining an intermediate language, perhaps based on three-address code or code for a stack machine, and writing a machine independent program to translate from the source code to this intermediate code. The intermediate code is then read by a second pass, and interpreted or compiled. This approach is elegant, and has a number of advantages, especially if the target machine is far removed from the host. It suffers from some disadvantages as well. Some constructions, like initialization and subroutine prologs, are difficult or expensive to express in a machine independent way that still allows them to be easily adapted to the target assemblers. Most of these approaches require a symbol table to be constructed in the second (machine dependent) pass, and/or require powerful target assemblers. Also, many conversion operators may be generated that have no effect on a given machine, but may be needed on others (for example, pointer to pointer conversions usually do nothing in C, but must be generated because there are some machines where they are significant).

For these reasons, the first pass of the portable compiler is not entirely machine independent. It contains some machine dependent features, such as initialization, subroutine prolog and epilog, certain storage allocation functions, code for the *switch* statement, and code to throw out unneeded conversion operators.

As a crude measure of the degree of portability actually achieved, the Interdata 8/32 C compiler has roughly 600 machine dependent lines of source out of 4600 in Pass 1, and 1000 out of 3400 in Pass 2. In total, 1600 out of 8000, or 20%, of the total source is machine dependent (12% in Pass 1, 30% in Pass 2). These percentages can be expected to rise slightly as the compiler is tuned. The percentage of machine-dependent code for the IBM is 22%, for the Honeywell 25%. If the assembler format and structure were the same for all these machines,

perhaps another 5-10% of the code would become machine independent.

These figures are sufficiently misleading as to be almost meaningless. A large fraction of the machine dependent code can be converted in a straightforward, almost mechanical way. On the other hand, a certain amount of the code requres hard intellectual effort to convert, since the algorithms embodied in this part of the code are typically complicated and machine dependent.

To summarize, however, if you need a C compiler written for a machine with a reasonable architecture, the compiler is already three quarters finished!

Overview

This paper discusses the structure and organization of the portable compiler. The intent is to give the big picture, rather than discussing the details of a particular machine implementation. After a brief overview and a discussion of the source file structure, the paper describes the major data structures, and then delves more closely into the two passes. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere.[4] One of the major design issues in any C compiler, the design of the calling sequence and stack frame, is the subject of a separate memorandum.[5]

The compiler consists of two passes, *pass1* and *pass2*, that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor, that handles the **#define** and **#include** statements, and related features (e.g., **#ifdef**, etc.). It is a nearly machine independent program, and will not be further discussed here.

The output of the preprocessor is a text file that is read as the standard input of the first pass. This produces as standard output another text file that becomes the standard input of the second pass. The second pass produces, as standard output, the desired assembler language source code. The preprocessor and the two passes all write error messages on the standard error file. Thus the compiler itself makes few demands on the I/O library support, aiding in the bootstrapping process.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program. This "one pass" operation eliminates the overhead of reading and writing the intermediate file, so the compiler operates about 30% faster in this mode. It also occupies about 30% more space than the larger of the two component passes.

Because the compiler is fundamentally structured as two passes, even when loaded as one, this document primarily describes the two pass version.

The first pass does the lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions, and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switches, and code for branches, label definitions, alignment operations, changes of location counter, etc.

The intermediate file is a text file organized into lines. Lines beginning with a right parenthesis are copied by the second pass directly to its output file, with the parenthesis stripped off. Thus, when the first pass produces assembly code, such as subroutine prologs, etc., each line is prefaced with a right parenthesis; the second pass passes these lines to through to the assembler.

The major job done by the second pass is generation of code for expressions. The expression parse trees produced in the first pass are written onto the intermediate file in Polish Prefix form: first, there is a line beginning with a period, followed by the source file line number and name on which the expression appeared (for debugging purposes). The successive lines represent the nodes of the parse tree, one node per line. Each line contains the node number, type, and any values (e.g., values of constants) that may appear in the node. Lines representing nodes with descendants are immediately followed by the left subtree of descendants, then the right. Since the number of descendants of any node is completely determined by the node

number, there is no need to mark the end of the tree.

There are only two other line types in the intermediate file. Lines beginning with a left square bracket ('[') represent the beginning of blocks (delimited by { ... } in the C source); lines beginning with right square brackets (']') represent the end of blocks. The remainder of these lines tell how much stack space, and how many register variables, are currently in use.

Thus, the second pass reads the intermediate files, copies the ')' lines, makes note of the information in the '[' and ']' lines, and devotes most of its effort to the '.' lines and their associated expression trees, turning them turns into assembly code to evaluate the expressions.

In the one pass version of the compiler, the expression trees that are built by the first pass have been declared to have room for the second pass information as well. Instead of writing the trees onto an intermediate file, each tree is transformed in place into an acceptable form for the code generator. The code generator then writes the result of compiling this tree onto the standard output. Instead of '[' and ']' lines in the intermediate file, the information is passed directly to the second pass routines. Assembly code produced by the first pass is simply written out, without the need for ')' at the head of each line.

The Source Files

The compiler source consists of 22 source files. Two files, *manifest* and *macdefs*, are header files included with all other files. *Manifest* has declarations for the node numbers, types, storage classes, and other global data definitions. *Macdefs* has machine-dependent definitions, such as the size and alignment of the various data representations. Two machine-independent header files, *mfile1* and *mfile2*, contain the data structure and manifest definitions for the first and second passes, respectively. In the second pass, a machine dependent header file, *mac2defs*, contains declarations of register names, etc.

There is a file, *common*, containing (machine independent) routines used in both passes. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages. There are two dummy files, *comm1.c* and *comm2.c*, that simply include *common* within the scope of the appropriate pass1 or pass2 header files. When the compiler is loaded as a single pass, *common* only needs to be included once: *comm2.c* is not needed.

Entire sections of this document are devoted to the detailed structure of the passes. For the moment, we just give a brief description of the files. The first pass is obtained by compiling and loading *scan.c*, *cgram.c*, *xdefs.c*, *pftn.c*, *trees.c*, *optim.c*, *local.c*, *code.c*, and *comm1.c*. *Scan.c* is the lexical analyzer, which is used by *cgram.c*, the result of applying *Yacc*[6] to the input grammar *cgram.y*. *Xdefs.c* is a short file of external definitions. *Pftn.c* maintains the symbol table, and does initialization. *Trees.c* builds the expression trees, and computes the node types. *Optim.c* does some machine independent optimizations on the expression trees. *Comm1.c* includes *common*, that contains service routines common to the two passes of the compiler. All the above files are machine independent. The files *local.c* and *code.c* contain machine dependent code for generating subroutine prologs, switch code, and the like.

The second pass is produced by compiling and loading *reader.c*, *allo.c*, *match.c*, *comm1.c*, *order.c*, *local.c*, and *table.c*. *Reader.c* reads the intermediate file, and controls the major logic of the code generation. *Allo.c* keeps track of busy and free registers. *Match.c* controls the matching of code templates to subtrees of the expression tree to be compiled. *Comm2.c* includes the file *common*, as in the first pass. The above files are machine independent. *Order.c* controls the machine dependent details of the code generation strategy. *Local2.c* has many small machine dependent routines, and tables of opcodes, register types, etc. *Table.c* has the code template tables, which are also clearly machine dependent.

Data Structure Considerations.

This section discusses the node numbers, type words, and expression trees, used throughout both passes of the compiler.

The file *manifest* defines those symbols used throughout both passes. The intent is to use the same symbol name (e.g., MINUS) for the given operator throughout the lexical analysis, parsing, tree building, and code generation phases; this requires some synchronization with the *Yacc* input file, *cgram.y*, as well.

A token like MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form (for example, $-=$), and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS.

It is frequently desirable to know if a node represents a leaf (no descendants), a unary operator (one descendant) or a binary operator (two descendants). The macro *optype(o)* returns one of the manifest constants LTYPE, UTYPE, or BITYPE, respectively, depending on the node number *o*. Similarly, *asgop(o)* returns true if *o* is an assignment operator number ($=$, $+=$, etc.), and *logop(o)* returns true if *o* is a relational or logical (&&, ||, or !) operator.

C has a rich typing structure, with a potentially infinite number of types. To begin with, there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Then, there are three operators that can be applied to types to make others: if *t* is a type, we may potentially have types *pointer to t*, *function returning t*, and *array of t's* generated from *t*. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators.

In the compiler, a type is represented by an unsigned integer; the rightmost four bits hold the basic type, and the remaining bits are divided into two-bit fields, containing 0 (no operator), or one of the three operators described above. The modifiers are read right to left in the word, starting with the two-bit field adjacent to the basic type, until a field with 0 in it is reached. The macros PTR, FTN, and ARY represent the *pointer to*, *function returning*, and *array of* operators. The macro values are shifted so that they align with the first two-bit field; thus PTR+INT represents the type for an integer pointer, and

$$\text{ARY} + (\text{PTR}<<2) + (\text{FTN}<<4) + \text{DOUBLE}$$

represents the type of an array of pointers to functions returning doubles.

The type words are ordinarily manipulated by macros. If *t* is a type word, *BTYPE(t)* gives the basic type. *ISPTR(t)*, *ISARY(t)*, and *ISFTN(t)* ask if an object of this type is a pointer, array, or a function, respectively. *MODTYPE(t,b)* sets the basic type of *t* to *b*. *DECREF(t)* gives the type resulting from removing the first operator from *t*. Thus, if *t* is a pointer to *t'*, a function returning *t'*, or an array of *t'*, then *DECREF(t)* would equal *t'*. *INCREF(t)* gives the type representing a pointer to *t*. Finally, there are operators for dealing with the unsigned types. *ISUNSIGNED(t)* returns true if *t* is one of the four basic unsigned types; in this case, *DEUNSIGN(t)* gives the associated 'signed' type. Similarly, *UNSIGNABLE(t)* returns true if *t* is one of the four basic types that could become unsigned, and *ENUNSIGN(t)* returns the unsigned analogue of *t* in this case.

The other important global data structure is that of expression trees. The actual shapes of the nodes are given in *mfile1* and *mfile2*. They are not the same in the two passes; the first pass nodes contain dimension and size information, while the second pass nodes contain register allocation information. Nevertheless, all nodes contain fields called *op*, containing the node number, and *type*, containing the type word. A function called *talloc()* returns a pointer to a new tree node. To free a node, its *op* field need merely be set to FREE. The other fields in the node will remain intact at least until the next allocation.

Nodes representing binary operators contain fields, *left* and *right*, that contain pointers to the left and right descendants. Unary operator nodes have the *left* field, and a value field called *rval*. Leaf nodes, with no descendants, have two value fields: *lval* and *rval*.

At appropriate times, the function *tcheck()* can be called, to check that there are no busy nodes remaining. This is used as a compiler consistency check. The function *tcopy(p)* takes a pointer *p* that points to an expression tree, and returns a pointer to a disjoint copy of the tree. The function *walkf(p,f)* performs a postorder walk of the tree pointed to by *p*, and applies the function *f* to each node. The function *fwalk(p,f,d)* does a preorder walk of the tree pointed to by *p*. At each node, it calls a function *f*, passing to it the node pointer, a value passed down from its ancestor, and two pointers to values to be passed down to the left and right descendants (if any). The value *d* is the value passed down to the root. *Fwalk* is used for a number of tree labeling and debugging activities.

The other major data structure, the symbol table, exists only in pass one, and will be discussed later.

Pass One

The first pass does lexical analysis, parsing, symbol table maintenance, tree building, optimization, and a number of machine dependent things. This pass is largely machine independent, and the machine independent sections can be pretty successfully ignored. Thus, they will be only sketched here.

Lexical Analysis

The lexical analyzer is a conceptually simple routine that reads the input and returns the tokens of the C language as it encounters them: names, constants, operators, and keywords. The conceptual simplicity of this job is confounded a bit by several other simple jobs that unfortunately must go on simultaneously. These include

- Keeping track of the current filename and line number, and occasionally setting this information as the result of preprocessor control lines.
- Skipping comments.
- Properly dealing with octal, decimal, hex, floating point, and character constants, as well as character strings.

To achieve speed, the program maintains several tables that are indexed into by character value, to tell the lexical analyzer what to do next. To achieve portability, these tables must be initialized each time the compiler is run, in order that the table entries reflect the local character set values.

Parsing

As mentioned above, the parser is generated by Yacc from the grammar on file *cgram.y*. The grammar is relatively readable, but contains some unusual features that are worth comment.

Perhaps the strangest feature of the grammar is the treatment of declarations. The problem is to keep track of the basic type and the storage class while interpreting the various stars, brackets, and parentheses that may surround a given name. The entire declaration mechanism must be recursive, since declarations may appear within declarations of structures and unions, or even within a sizeof construction inside a dimension in another declaration!

There are some difficulties in using a bottom-up parser, such as produced by Yacc, to handle constructions where a lot of left context information must be kept around. The problem is that the original PDP-11 compiler is top-down in implementation, and some of the semantics of C reflect this. In a top-down parser, the input rules are restricted somewhat, but one can naturally associate temporary storage with a rule at a very early stage in the recognition of that rule. In a bottom-up parser, there is more freedom in the specification of rules, but it is more

difficult to know what rule is being matched until the entire rule is seen. The parser described by *cgram.c* makes effective use of the bottom-up parsing mechanism in some places (notably the treatment of expressions), but struggles against the restrictions in others. The usual result is that it is necessary to run a stack of values "on the side", independent of the Yacc value stack, in order to be able to store and access information deep within inner constructions, where the relationship of the rules being recognized to the total picture is not yet clear.

In the case of declarations, the attribute information (type, etc.) for a declaration is carefully kept immediately to the left of the declarator (that part of the declaration involving the name). In this way, when it is time to declare the name, the name and the type information can be quickly brought together. The "$0" mechanism of Yacc is used to accomplish this. The result is not pretty, but it works. The storage class information changes more slowly, so it is kept in an external variable, and stacked if necessary. Some of the grammar could be considerably cleaned up by using some more recent features of Yacc, notably actions within rules and the ability to return multiple values for actions.

A stack is also used to keep track of the current location to be branched to when a **break** or **continue** statement is processed.

This use of external stacks dates from the time when Yacc did not permit values to be structures. Some, or most, of this use of external stacks could be eliminated by redoing the grammar to use the mechanisms now provided. There are some areas, however, particularly the processing of structure, union, and enum declarations, function prologs, and switch statement processing, when having all the affected data together in an array speeds later processing; in this case, use of external storage seems essential.

The *cgram.y* file also contains some small functions used as utility functions in the parser. These include routines for saving case values and labels in processing switches, and stacking and popping values on the external stack described above.

Storage Classes

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass; by the second pass, this information must have been totally dealt with. This means that all of the storage allocation must take place in the first pass, so that references to automatics and parameters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. Much of this transformation is machine dependent, and strongly depends on the storage class.

The classes include EXTERN (for externally declared, but not defined variables), EXTDEF (for external definitions), and similar distinctions for USTATIC and STATIC, UFORTRAN and FORTRAN (for fortran functions) and ULABEL and LABEL. The storage classes REGISTER and AUTO are obvious, as are STNAME, UNAME, and ENAME (for structure, union, and enumeration tags), and the associated MOS, MOU, and MOE (for the members). TYPEDEF is treated as a storage class as well. There are two special storage classes: PARAM and SNULL. SNULL is used to distinguish the case where no explicit storage class has been given; before an entry is made in the symbol table the true storage class is discovered. Similarly, PARAM is used for the temporary entry in the symbol table made before the declaration of function parameters is completed.

The most complexity in the storage class process comes from bit fields. A separate storage class is kept for each width bit field; a k bit bit field has storage class k plus FIELD. This enables the size to be quickly recovered from the storage class.

Symbol Table Maintenance.

The symbol table routines do far more than simply enter names into the symbol table; considerable semantic processing and checking is done as well. For example, if a new declaration comes in, it must be checked to see if there is a previous declaration of the same symbol. If there is, there are many cases. The declarations may agree and be compatible (for example, an extern declaration can appear twice) in which case the new declaration is ignored. The new declaration may add information (such as an explicit array dimension) to an already present declaration. The new declaration may be different, but still correct (for example, an extern declaration of something may be entered, and then later the definition may be seen). The new declaration may be incompatible, but appear in an inner block; in this case, the old declaration is carefully hidden away, and the new one comes into force until the block is left. Finally, the declarations may be incompatible, and an error message must be produced.

A number of other factors make for additional complexity. The type declared by the user is not always the type entered into the symbol table (for example, if an formal parameter to a function is declared to be an array, C requires that this be changed into a pointer before entry in the symbol table). Moreover, there are various kinds of illegal types that may be declared which are difficult to check for syntactically (for example, a function returning an array). Finally, there is a strange feature in C that requires structure tag names and member names for structures and unions to be taken from a different logical symbol table than ordinary identifiers. Keeping track of which kind of name is involved is a bit of struggle (consider typedef names used within structure declarations, for example).

The symbol table handling routines have been rewritten a number of times to extend features, improve performance, and fix bugs. They address the above problems with reasonable effectiveness but a singular lack of grace.

When a name is read in the input, it is hashed, and the routine *lookup* is called, together with a flag which tells which symbol table should be searched (actually, both symbol tables are stored in one, and a flag is used to distinguish individual entries). If the name is found, *lookup* returns the index to the entry found; otherwise, it makes a new entry, marks it UNDEF (undefined), and returns the index of the new entry. This index is stored in the *rval* field of a NAME node.

When a declaration is being parsed, this NAME node is made part of a tree with UNARY MUL nodes for each *, LB nodes for each array descriptor (the right descendant has the dimension), and UNARY CALL nodes for each function descriptor. This tree is passed to the routine *tymerge*, along with the attribute type of the whole declaration; this routine collapses the tree to a single node, by calling *tyreduce*, and then modifies the type to reflect the overall type of the declaration.

Dimension and size information is stored in a table called *dimtab*. To properly describe a type in C, one needs not just the type information but also size information (for structures and enums) and dimension information (for arrays). Sizes and offsets are dealt with in the compiler by giving the associated indices into *dimtab*. *Tymerge* and *tyreduce* call *dstash* to put the discovered dimensions away into the *dimtab* array. *Tymerge* returns a pointer to a single node that contains the symbol table index in its *rval* field, and the size and dimension indices in fields *csiz* and *cdim*, respectively. This information is properly considered part of the type in the first pass, and is carried around at all times.

To enter an element into the symbol table, the routine *defid* is called; it is handed a storage class, and a pointer to the node produced by *tymerge*. *Defid* calls *fixtype*, which adjusts and checks the given type depending on the storage class, and converts null types appropriately. It then calls *fixclass*, which does a similar job for the storage class; it is here, for example, that register declarations are either allowed or changed to auto.

The new declaration is now compared against an older one, if present, and several pages of validity checks performed. If the definitions are compatible, with possibly some added information, the processing is straightforward. If the definitions differ, the block levels of the

current and the old declaration are compared. The current block level is kept in *blevel*, an external variable; the old declaration level is kept in the symbol table. Block level 0 is for external declarations, 1 is for arguments to functions, and 2 and above are blocks within a function. If the current block level is the same as the old declaration, an error results. If the current block level is higher, the new declaration overrides the old. This is done by marking the old symbol table entry "hidden", and making a new entry, marked "hiding". *Lookup* will skip over hidden entries. When a block is left, the symbol table is searched, and any entries defined in that block are destroyed; if they hid other entries, the old entries are "unhidden".

This nice block structure is warped a bit because labels do not follow the block structure rules (one can do a goto into a block, for example); default definitions of functions in inner blocks also persist clear out to the outermost scope. This implies that cleaning up the symbol table after block exit is more subtle than it might first seem.

For successful new definitions, *defid* also initializes a "general purpose" field, *offset*, in the symbol table. It contains the stack offset for automatics and parameters, the register number for register variables, the bit offset into the structure for structure members, and the internal label number for static variables and labels. The offset field is set by *falloc* for bit fields, and *dclstruct* for structures and unions.

The symbol table entry itself thus contains the name, type word, size and dimension offsets, offset value, and declaration block level. It also has a field of flags, describing what symbol table the name is in, and whether the entry is hidden, or hides another. Finally, a field gives the line number of the last use, or of the definition, of the name. This is used mainly for diagnostics, but is useful to *lint* as well.

In some special cases, there is more than the above amount of information kept for the use of the compiler. This is especially true with structures; for use in initialization, structure declarations must have access to a list of the members of the structure. This list is also kept in *dimtab*. Because a structure can be mentioned long before the members are known, it is necessary to have another level of indirection in the table. The two words following the *csiz* entry in *dimtab* are used to hold the alignment of the structure, and the index in dimtab of the list of members. This list contains the symbol table indices for the structure members, terminated by a -1.

Tree Building

The portable compiler transforms expressions into expression trees. As the parser recognizes each rule making up an expression, it calls *buildtree* which is given an operator number, and pointers to the left and right descendants. *Buildtree* first examines the left and right descendants, and, if they are both constants, and the operator is appropriate, simply does the constant computation at compile time, and returns the result as a constant. Otherwise, *buildtree* allocates a node for the head of the tree, attaches the descendants to it, and ensures that conversion operators are generated if needed, and that the type of the new node is consistent with the types of the operands. There is also a considerable amount of semantic complexity here; many combinations of types are illegal, and the portable compiler makes a strong effort to check the legality of expression types completely. This is done both for *lint* purposes, and to prevent such semantic errors from being passed through to the code generator.

The heart of *buildtree* is a large table, accessed by the routine *opact*. This routine maps the types of the left and right operands into a rather smaller set of descriptors, and then accesses a table (actually encoded in a switch statement) which for each operator and pair of types causes an action to be returned. The actions are logical or's of a number of separate actions, which may be carried out by *buildtree*. These component actions may include checking the left side to ensure that it is an lvalue (can be stored into), applying a type conversion to the left or right operand, setting the type of the new node to the type of the left or right operand, calling various routines to balance the types of the left and right operands, and suppressing the ordinary conversion of arrays and function operands to pointers. An important operation is OTHER, which causes some special code to be invoked in *buildtree*, to handle issues which are

unique to a particular operator. Examples of this are structure and union reference (actually handled by the routine *stref*), the building of NAME, ICON, STRING and FCON (floating point constant) nodes, unary * and &, structure assignment, and calls. In the case of unary * and &, *buildtree* will cancel a * applied to a tree, the top node of which is &, and conversely.

Another special operation is PUN; this causes the compiler to check for type mismatches, such as intermixing pointers and integers.

The treatment of conversion operators is still a rather strange area of the compiler (and of C!). The recent introduction of type casts has only confounded this situation. Most of the conversion operators are generated by calls to *tymatch* and *ptmatch*, both of which are given a tree, and asked to make the operands agree in type. *Ptmatch* treats the case where one of the operands is a pointer; *tymatch* treats all other cases. Where these routines have decided on the proper type for an operand, they call *makety*, which is handed a tree, and a type word, dimension offset, and size offset. If necessary, it inserts a conversion operation to make the types correct. Conversion operations are never inserted on the left side of assignment operators, however. There are two conversion operators used; PCONV, if the conversion is to a non-basic type (usually a pointer), and SCONV, if the conversion is to a basic type (scalar).

To allow for maximum flexibility, every node produced by *buildtree* is given to a machine dependent routine, *clocal*, immediately after it is produced. This is to allow more or less immediate rewriting of those nodes which must be adapted for the local machine. The conversion operations are given to *clocal* as well; on most machines, many of these conversions do nothing, and should be thrown away (being careful to retain the type). If this operation is done too early, however, later calls to *buildtree* may get confused about correct type of the subtrees; thus *clocal* is given the conversion ops only after the entire tree is built. This topic will be dealt with in more detail later.

Initialization

Initialization is one of the messier areas in the portable compiler. The only consolation is that most of the mess takes place in the machine independent part, where it is may be safely ignored by the implementor of the compiler for a particular machine.

The basic problem is that the semantics of initialization really calls for a co-routine structure; one collection of programs reading constants from the input stream, while another, independent set of programs places these constants into the appropriate spots in memory. The dramatic differences in the local assemblers also come to the fore here. The parsing problems are dealt with by keeping a rather extensive stack containing the current state of the initialization; the assembler problems are dealt with by having a fair number of machine dependent routines.

The stack contains the symbol table number, type, dimension index, and size index for the current identifier being initialized. Another entry has the offset, in bits, of the beginning of the current identifier. Another entry keeps track of how many elements have been seen, if the current identifier is an array. Still another entry keeps track of the current member of a structure being initialized. Finally, there is an entry containing flags which keep track of the current state of the initialization process (e.g., tell if a } has been seen for the current identifier.)

When an initialization begins, the routine *beginit* is called; it handles the alignment restrictions, if any, and calls *instk* to create the stack entry. This is done by first making an entry on the top of the stack for the item being initialized. If the top entry is an array, another entry is made on the stack for the first element. If the top entry is a structure, another entry is made on the stack for the first member of the structure. This continues until the top element of the stack is a scalar. *Instk* then returns, and the parser begins collecting initializers.

When a constant is obtained, the routine *doinit* is called; it examines the stack, and does whatever is necessary to assign the current constant to the scalar on the top of the stack. *gotscal* is then called, which rearranges the stack so that the next scalar to be initialized gets placed on top of the stack. This process continues until the end of the initializers; *endinit* cleans up. If

a { or } is encountered in the string of initializers, it is handled by calling *ilbrace* or *irbrace*, respectively.

A central issue is the treatment of the "holes" that arise as a result of alignment restrictions or explicit requests for holes in bit fields. There is a global variable, *inoff*, which contains the current offset in the initialization (all offsets in the first pass of the compiler are in bits). *Doinit* figures out from the top entry on the stack the expected bit offset of the next identifier; it calls the machine dependent routine *inforce* which, in a machine dependent way, forces the assembler to set aside space if need be so that the next scalar seen will go into the appropriate bit offset position. The scalar itself is passed to one of the machine dependent routines *fincode* (for floating point initialization), *incode* (for fields, and other initializations less than an int in size), and *cinit* (for all other initializations). The size is passed to all these routines, and it is up to the machine dependent routines to ensure that the initializer occupies exactly the right size.

Character strings represent a bit of an exception. If a character string is seen as the initializer for a pointer, the characters making up the string must be put out under a different location counter. When the lexical analyzer sees the quote at the head of a character string, it returns the token STRING, but does not do anything with the contents. The parser calls *getstr*, which sets up the appropriate location counters and flags, and calls *lxstr* to read and process the contents of the string.

If the string is being used to initialize a character array, *lxstr* calls *putbyte*, which in effect simulates *doinit* for each character read. If the string is used to initialize a character pointer, *lxstr* calls a machine dependent routine, *bycode*, which stashes away each character. The pointer to this string is then returned, and processed normally by *doinit*.

The null at the end of the string is treated as if it were read explicitly by *lxstr*.

Statements

The first pass addresses four main areas; declarations, expressions, initialization, and statements. The statement processing is relatively simple; most of it is carried out in the parser directly. Most of the logic is concerned with allocating label numbers, defining the labels, and branching appropriately. An external symbol, *reached*, is 1 if a statement can be reached, 0 otherwise; this is used to do a bit of simple flow analysis as the program is being parsed, and also to avoid generating the subroutine return sequence if the subroutine cannot "fall through" the last statement.

Conditional branches are handled by generating an expression node, CBRANCH, whose left descendant is the conditional expression and the right descendant is an ICON node containing the internal label number to be branched to. For efficiency, the semantics are that the label is gone to if the condition is *false*.

The switch statement is compiled by collecting the case entries, and an indication as to whether there is a default case; an internal label number is generated for each of these, and remembered in a big array. The expression comprising the value to be switched on is compiled when the switch keyword is encountered, but the expression tree is headed by a special node, FORCE, which tells the code generator to put the expression value into a special distinguished register (this same mechanism is used for processing the return statement). When the end of the switch block is reached, the array containing the case values is sorted, and checked for duplicate entries (an error); if all is correct, the machine dependent routine *genswitch* is called, with this array of labels and values in increasing order. *Genswitch* can assume that the value to be tested is already in the register which is the usual integer return value register.

Optimization

There is a machine independent file, *optim.c*, which contains a relatively short optimization routine, *optim*. Actually the word optimization is something of a misnomer; the results are not optimum, only improved, and the routine is in fact not optional; it must be called for proper operation of the compiler.

Optim is called after an expression tree is built, but before the code generator is called. The essential part of its job is to call *clocal* on the conversion operators. On most machines, the treatment of & is also essential: by this time in the processing, the only node which is a legal descendant of & is NAME. (Possible descendants of * have been eliminated by *buildtree.)* The address of a static name is, almost by definition, a constant, and can be represented by an ICON node on most machines (provided that the loader has enough power). Unfortunately, this is not universally true; on some machines, such as the IBM 370, the issue of addressability rears its ugly head; thus, before turning a NAME node into an ICON node, the machine dependent function *andable* is called.

The optimization attempts of *optim* are currently quite limited. It is primarily concerned with improving the behavior of the compiler with operations one of whose arguments is a constant. In the simplest case, the constant is placed on the right if the operation is commutative. The compiler also makes a limited search for expressions such as

$$(x + a) + b$$

where a and b are constants, and attempts to combine a and b at compile time. A number of special cases are also examined; additions of 0 and multiplications by 1 are removed, although the correct processing of these cases to get the type of the resulting tree correct is decidedly nontrivial. In some cases, the addition or multiplication must be replaced by a conversion op to keep the types from becoming fouled up. Finally, in cases where a relational operation is being done, and one operand is a constant, the operands are permuted, and the operator altered, if necessary, to put the constant on the right. Finally, multiplications by a power of 2 are changed to shifts.

There are dozens of similar optimizations that can be, and should be, done. It seems likely that this routine will be expanded in the relatively near future.

Machine Dependent Stuff

A number of the first pass machine dependent routines have been discussed above. In general, the routines are short, and easy to adapt from machine to machine. The two exceptions to this general rule are *clocal* and the function prolog and epilog generation routines, *bfcode* and *efcode.*

Clocal has the job of rewriting, if appropriate and desirable, the nodes constructed by *buildtree.* There are two major areas where this is important; NAME nodes and conversion operations. In the case of NAME nodes, *clocal* must rewrite the NAME node to reflect the actual physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through the *rval* field of the node), and, based on the storage class of the node, the tree must be rewritten. Automatic variables and parameters are typically rewritten by treating the reference to the variable as a structure reference, off the register which holds the stack or argument pointer; the *stref* routine is set up to be called in this way, and to build the appropriate tree. In the most general case, the tree consists of a unary * node, whose descendant is a + node, with the stack or argument register as left operand, and a constant offset as right operand. In the case of LABEL and internal static nodes, the *rval* field is rewritten to be the negative of the internal label number; a negative *rval* field is taken to be an internal label number. Finally, a name of class REGISTER must be converted into a REG node, and the *rval* field replaced by the register number. In fact, this part of the *clocal* routine is nearly machine independent; only for machines with addressability problems (IBM 370 again!) does it have to be noticeably different,

The conversion operator treatment is rather tricky. It is necessary to handle the application of conversion operators to constants in *clocal,* in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversion from byte pointer to short or long pointer might require a shift or rotate

operation, which would have to be generated here.

The extension of the portable compiler to machines where the size of a pointer depends on its type would be straightforward, but has not yet been done.

The other major machine dependent issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence; this design issue is discussed elsewhere.[5] The routine *bfcode* is called with the number of arguments the function is defined with, and an array containing the symbol table indices of the declared parameters. *Bfcode* must generate the code to establish the new stack frame, save the return address and previous stack pointer value on the stack, and save whatever registers are to be used for register variables. The stack size and the number of register variables is not known when *bfcode* is called, so these numbers must be referred to by assembler constants, which are defined when they are known (usually in the second pass, after all register variables, automatics, and temporaries have been seen). The final job is to find those parameters which may have been declared register, and generate the code to initialize the register with the value passed on the stack. Once again, for most machines, the general logic of *bfcode* remains the same, but the contents of the *printf* calls in it will change from machine to machine. *efcode* is rather simpler, having just to generate the default return at the end of a function. This may be nontrivial in the case of a function returning a structure or union, however.

There seems to be no really good place to discuss structures and unions, but this is as good a place as any. The C language now supports structure assignment, and the passing of structures as arguments to functions, and the receiving of structures back from functions. This was added rather late to C, and thus to the portable compiler. Consequently, it fits in less well than the older features. Moreover, most of the burden of making these features work is placed on the machine dependent code.

There are both conceptual and practical problems. Conceptually, the compiler is structured around the idea that to compute something, you put it into a register and work on it. This notion causes a bit of trouble on some machines (e.g., machines with 3-address opcodes), but matches many machines quite well. Unfortunately, this notion breaks down with structures. The closest that one can come is to keep the addresses of the structures in registers. The actual code sequences used to move structures vary from the trivial (a multiple byte move) to the horrible (a function call), and are very machine dependent.

The practical problem is more painful. When a function returning a structure is called, this function has to have some place to put the structure value. If it places it on the stack, it has difficulty popping its stack frame. If it places the value in a static temporary, the routine fails to be reentrant. The most logically consistent way of implementing this is for the caller to pass in a pointer to a spot where the called function should put the value before returning. This is relatively straightforward, although a bit tedious, to implement, but means that the caller must have properly declared the function type, even if the value is never used. On some machines, such as the Interdata 8/32, the return value simply overlays the argument region (which on the 8/32 is part of the caller's stack frame). The caller takes care of leaving enough room if the returned value is larger than the arguments. This also assumes that the caller know and declares the function properly.

The PDP-11 and the VAX have stack hardware which is used in function calls and returns; this makes it very inconvenient to use either of the above mechanisms. In these machines, a static area within the called functionis allocated, and the function return value is copied into it on return; the function returns the address of that region. This is simple to implement, but is non-reentrant. However, the function can now be called as a subroutine without being properly declared, without the disaster which would otherwise ensue. No matter what choice is taken, the convention is that the function actually returns the address of the return structure value.

In building expression trees, the portable compiler takes a bit for granted about structures. It assumes that functions returning structures actually return a pointer to the structure, and it

assumes that a reference to a structure is actually a reference to its address. The structure assignment operator is rebuilt so that the left operand is the structure being assigned to, but the right operand is the address of the structure being assigned; this makes it easier to deal with

$$a = b = c$$

and similar constructions.

There are four special tree nodes associated with these operations: STASG (structure assignment), STARG (structure argument to a function call), and STCALL and UNARY STCALL (calls of a function with nonzero and zero arguments, respectively). These four nodes are unique in that the size and alignment information, which can be determined by the type for all other objects in C, must be known to carry out these operations; special fields are set aside in these nodes to contain this information, and special intermediate code is used to transmit this information.

First Pass Summary

There are may other issues which have been ignored here, partly to justify the title "tour", and partially because they have seemed to cause little trouble. There are some debugging flags which may be turned on, by giving the compiler's first pass the argument

−X[flags]

Some of the more interesting flags are −Xd for the defining and freeing of symbols, −Xi for initialization comments, and −Xb for various comments about the building of trees. In many cases, repeating the flag more than once gives more information; thus, −Xddd gives more information than −Xd. In the two pass version of the compiler, the flags should not be set when the output is sent to the second pass, since the debugging output and the intermediate code both go onto the standard output.

We turn now to consideration of the second pass.

Pass Two

Code generation is far less well understood than parsing or lexical analysis, and for this reason the second pass is far harder to discuss in a file by file manner. A great deal of the difficulty is in understanding the issues and the strategies employed to meet them. Any particular function is likely to be reasonably straightforward.

Thus, this part of the paper will concentrate a good deal on the broader aspects of strategy in the code generator, and will not get too intimate with the details.

Overview.

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines, and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code, rather than to produce incorrect code. This goal is achieved by having a simple model of the job to be done (e.g., an expression tree) and a simple model of the machine state (e.g., which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state; hopefully, the tree eventually gets reduced to a single node. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all the C operators. Thus the selection of which instruction/transformations to generate, and in what order, will have a heuristic flavor. If, for some expression tree, no transformation applies, or, more seriously, if the heuristics select a sequence of instruction/transformations that do not in fact reduce the tree, the compiler will report its inability to generate code, and abort.

A major part of the code generator is concerned with the model and the transformations, — most of this is machine independent, or depends only on simple tables. The flexibility comes from the heuristics that guide the transformations of the trees, the selection of subgoals, and the ordering of the computation.

The Machine Model

The machine is assumed to have a number of registers, of at most two different types: *A* and *B*. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g., register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number in the *mac2defs* file; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table is the *rstatus* table on file *local2.c*. This table is indexed by register number, and contains expressions made up from manifest constants describing the register types: SAREG for dedicated AREG's, SAREGISTAREG for scratch AREGS's, and SBREG and SBREGISTBREG similarly for BREG's. There are macros that access this information: *isbreg(r)* returns true if register number *r* is a BREG, and *isreg(r)* returns true if register number *r* is a temporary AREG or BREG. Another table, *rnames*, contains the register names; this is used when putting out assembler code and diagnostics.

The usage of registers is kept track of by an array called *busy*. *Busy[r]* is the number of uses of register *r* in the current tree being processed. The allocation and freeing of registers will be discussed later as part of the code generation algorithm.

General Organization

As mentioned above, the second pass reads lines from the intermediate file, copying through to the output unchanged any lines that begin with a ')', and making note of the information about stack usage and register allocation contained on lines beginning with ']' and '['. The expression trees, whose beginning is indicated by a line beginning with '.', are read and rebuilt into trees. If the compiler is loaded as one pass, the expression trees are immediately available to the code generator.

The actual code generation is done by a hierarchy of routines. The routine *delay* is first given the tree; it attempts to delay some postfix ++ and −− computations that might reasonably be done after the smoke clears. It also attempts to handle comma (,) operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. *Delay* calls *codgen* to control the actual code generation process. *Codgen* takes as arguments a pointer to the expression tree, and a second argument that, for socio-historical reasons, is called a *cookie*. The cookie describes a set of goals that would be acceptable for the code generation: these are assigned to individual bits, so they may be logically or'ed together to form a large number of possible goals. Among the possible goals are FOREFF (compute for side effects only; don't worry about the value), INTEMP (compute and store value into a temporary location in memory), INAREG (compute into an A register), INTAREG (compute into a scratch A register), INBREG and INTBREG similarly, FORCC (compute for condition codes), and FORARG (compute it as a function argument; e.g., stack it if appropriate).

Codgen first canonicalizes the tree by calling *canon*. This routine looks for certain transformations that might now be applicable to the tree. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its

behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register. *Canon* also looks for such cases, calling the machine dependent routine *notoff* to decide if the offset is acceptable (for example, in the IBM 370 the offset must be between 0 and 4095 bytes). Another optimization is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine, *sucomp*, is called that computes the Sethi-Ullman numbers for the tree (see below).

After the tree is canonicalized, *codgen* calls the routine *store* whose job is to select a subtree of the tree to be computed and (usually) stored before beginning the computation of the full tree. *Store* must return a tree that can be computed without need for any temporary storage locations. In effect, the only store operations generated while processing the subtree must be as a response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator can operate under the assumption that there are enough registers to do its job, without worrying about temporary storage. If a store into a temporary appears in the output, it is always as a direct result of logic in the *store* routine; this makes debugging easier.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision.[7] It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

The *store* routine decides what subtrees, if any, should be stored by making use of numbers, called *Sethi-Ullman numbers*, that give, for each subtree of an expression tree, the minimum number of scratch registers required to compile the subtree, without any stores into temporaries.[8] These numbers are computed by the machine-dependent routine *sucomp*, called by *canon*. The basic notion is that, knowing the Sethi-Ullman numbers for the descendants of a node, and knowing the operator of the node and some information about the machine, the Sethi-Ullman number of the node itself can be computed. If the Sethi-Ullman number for a tree exceeds the number of scratch registers available, some subtree must be stored. Unfortunately, the theory behind the Sethi-Ullman numbers applies only to uselessly simple machines and operators. For the rich set of C operators, and for machines with asymmetric registers, register pairs, different kinds of registers, and exceptional forms of addressing, the theory cannot be applied directly. The basic idea of estimation is a good one, however, and well worth applying; the application, especially when the compiler comes to be tuned for high code quality, goes beyond the park of theory into the swamp of heuristics. This topic will be taken up again later, when more of the compiler structure has been described.

After examining the Sethi-Ullman numbers, *store* selects a subtree, if any, to be stored, and returns the subtree and the associated cookie in the external variables *stotree* and *stocook*. If a subtree has been selected, or if the whole tree is ready to be processed, the routine *order* is called, with a tree and cookie. *Order* generates code for trees that do not require temporary locations. *Order* may make recursive calls on itself, and, in some cases, on *codgen*; for example, when processing the operators &&, ||, and comma (','), that have a left to right evaluation, it is incorrect for *store* examine the right operand for subtrees to be stored. In these cases, *order* will call *codgen* recursively when it is permissible to work on the right operand. A similar issue arises with the ? : operator.

The *order* routine works by matching the current tree with a set of code templates. If a template is discovered that will match the current tree and cookie, the associated assembly language statement or statements are generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to find a match with a different cookie; for example, in order to compute an expression with cookie INTEMP (store into a temporary storage location), it is usually necessary to compute the expression into a scratch register first. If all attempts to match the

tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call *order* to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate. For example, if the left operand of a + is a scratch register, the + can be replaced by a += operator; the tree may then match a template.

To close this introduction, we will discuss the steps in compiling code for the expression

a += *b*

where *a* and *b* are static variables.

To begin with, the whole expression tree is examined with cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the Interdata 8/32 for the moment. It is recognized that the left hand and right hand sides of the += operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite this as

a = *a* + *b*

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it is recognized that the left hand side of the assignment is addressable, but the right hand side is not in a register, so *order* is called recursively, being asked to put the right hand side of the assignment into a register. This invocation of *order* searches the tree for a match, and fails. The machine dependent rule for + notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call to *order* is made, with the tree consisting solely of the leaf *a*, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted. The node consisting of *a* is rewritten in place to represent the register into which *a* is loaded, and this third call to *order* returns. The second call to *order* now finds that it has the tree

reg + *b*

to consider. Once again, there is no match, but the default rewriting rule rewrites the + as a += operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

reg += *b*

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of *order*, so this invocation terminates, returning to the first level. The original tree has now become

a = **reg**

which matches a template for the store instruction. The store is output, and the tree rewritten to become just a single register node. At this point, since the top level call to *order* was interested only in side effects, the call to *order* returns, and the code generation is completed; we have generated a load, add, and store, as might have been expected.

The effect of machine architecture on this is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different; *b* is loaded in to a register, and then an add to storage instruction generated to add this register in to *a*. The transformations, involving as they do the semantics of C, are largely machine independent. The decisions as to when to use them, however, are almost totally machine dependent.

Having given a broad outline of the code generation process, we shall next consider the

heart of it: the templates. This leads naturally into discussions of template matching and register allocation, and finally a discussion of the machine dependent interfaces and strategies.

The Templates

The templates describe the effect of the target machine instructions on the model of computation around which the compiler is organized. In effect, each template has five logical sections, and represents an assertion of the form:

> **If** we have a subtree of a given shape (1), and we have a goal (cookie) or goals to achieve (2), and we have sufficient free resources (3), **then** we may emit an instruction or instructions (4), and rewrite the subtree in a particular manner (5), and the rewritten tree will achieve the desired goals.

These five sections will be discussed in more detail later. First, we give an example of a template:

```
ASG PLUS,   INAREG,
            SAREG,      TINT,
            SNAME,      TINT,
                        0,          RLEFT,
                        "           add         AL,AR\n",
```

The top line specifies the operator (+=) and the cookie (compute the value of the subtree into an AREG). The second and third lines specify the left and right descendants, respectively, of the += operator. The left descendant must be a REG node, representing an A register, and have integer type, while the right side must be a NAME node, and also have integer type. The fourth line contains the resource requirements (no scratch registers or temporaries needed), and the rewriting rule (replace the subtree by the left descendant). Finally, the quoted string on the last line represents the output to the assembler: lower case letters, tabs, spaces, etc. are copied *verbatim* to the output; upper case letters trigger various macro-like expansions. Thus, **AL** would expand into the Address form of the Left operand — presumably the register number. Similarly, **AR** would expand into the name of the right operand. The *add* instruction of the last section might well be emitted by this template.

In principle, it would be possible to make separate templates for all legal combinations of operators, cookies, types, and shapes. In practice, the number of combinations is very large. Thus, a considerable amount of mechanism is present to permit a large number of subtrees to be matched by a single template. Most of the shape and type specifiers are individual bits, and can be logically or'ed together. There are a number of special descriptors for matching classes of operators. The cookies can also be combined. As an example of the kind of template that really arises in practice, the actual template for the Interdata 8/32 that subsumes the above example is:

```
ASG OPSIMP, INAREG|FORCC,
            SAREG,      TINT|TUNSIGNED|TPOINT,
            SAREG|SNAME|SOREG|SCON,     TINT|TUNSIGNED|TPOINT,
                        0,          RLEFT|RESCC,
                        "           OI          AL,AR\n",
```

Here, OPSIMP represents the operators +, −, |, &, and ^. The **OI** macro in the output string expands into the appropriate Integer Opcode for the operator. The left and right sides can be integers, unsigned, or pointer types. The right side can be, in addition to a name, a register, a memory location whose address is given by a register and displacement (OREG), or a constant. Finally, these instructions set the condition codes, and so can be used in condition contexts: the cookie and rewriting rules reflect this.

The Template Matching Algorithm.

The heart of the second pass is the template matching algorithm, in the routine *match*. *Match* is called with a tree and a cookie; it attempts to match the given tree against some template that will transform it according to one of the goals given in the cookie. If a match is successful, the transformation is applied; *expand* is called to generate the assembly code, and then *reclaim* rewrites the tree, and reclaims the resources, such as registers, that might have become free as a result of the generated code.

This part of the compiler is among the most time critical. There is a spectrum of implementation techniques available for doing this matching. The most naive algorithm simply looks at the templates one by one. This can be considerably improved upon by restricting the search for an acceptable template. It would be possible to do better than this if the templates were given to a separate program that ate them and generated a template matching subroutine. This would make maintenance of the compiler much more complicated, however, so this has not been done.

The matching algorithm is actually carried out by restricting the range in the table that must be searched for each opcode. This introduces a number of complications, however, and needs a bit of sympathetic help by the person constructing the compiler in order to obtain best results. The exact tuning of this algorithm continues; it is best to consult the code and comments in *match* for the latest version.

In order to match a template to a tree, it is necessary to match not only the cookie and the op of the root, but also the types and shapes of the left and right descendants (if any) of the tree. A convention is established here that is carried out throughout the second pass of the compiler. If a node represents a unary operator, the single descendant is always the "left" descendant. If a node represents a unary operator or a leaf node (no descendants) the "right" descendant is taken by convention to be the node itself. This enables templates to easily match leaves and conversion operators, for example, without any additional mechanism in the matching program.

The type matching is straightforward; it is possible to specify any combination of basic types, general pointers, and pointers to one or more of the basic types. The shape matching is somewhat more complicated, but still pretty simple. Templates have a collection of possible operand shapes on which the opcode might match. In the simplest case, an *add* operation might be able to add to either a register variable or a scratch register, and might be able (with appropriate help from the assembler) to add an integer constant (ICON), a static memory cell (NAME), or a stack location (OREG).

It is usually attractive to specify a number of such shapes, and distinguish between them when the assembler output is produced. It is possible to describe the union of many elementary shapes such as ICON, NAME, OREG, AREG or BREG (both scratch and register forms), etc. To handle at least the simple forms of indirection, one can also match some more complicated forms of trees; STARNM and STARREG can match more complicated trees headed by an indirection operator, and SFLD can match certain trees headed by a FLD operator: these patterns call machine dependent routines that match the patterns of interest on a given machine. The shape SWADD may be used to recognize NAME or OREG nodes that lie on word boundaries: this may be of some importance on word−addressed machines. Finally, there are some special shapes: these may not be used in conjunction with the other shapes, but may be defined and extended in machine dependent ways. The special shapes SZERO, SONE, and SMONE are predefined and match constants 0, 1, and −1, respectively; others are easy to add and match by using the machine dependent routine *special*.

When a template has been found that matches the root of the tree, the cookie, and the shapes and types of the descendants, there is still one bar to a total match: the template may call for some resources (for example, a scratch register). The routine *allo* is called, and it attempts to allocate the resources. If it cannot, the match fails; no resources are allocated. If successful, the allocated resources are given numbers 1, 2, etc. for later reference when the

assembly code is generated. The routines *expand* and *reclaim* are then called. The *match* routine then returns a special value, MDONE. If no match was found, the value MNOPE is returned; this is a signal to the caller to try more cookie values, or attempt a rewriting rule. *Match* is also used to select rewriting rules, although the way of doing this is pretty straightforward. A special cookie, FORREW, is used to ask *match* to search for a rewriting rule. The rewriting rules are keyed to various opcodes; most are carried out in *order*. Since the question of when to rewrite is one of the key issues in code generation, it will be taken up again later.

Register Allocation.

The register allocation routines, and the allocation strategy, play a central role in the correctness of the code generation algorithm. If there are bugs in the Sethi-Ullman computation that cause the number of needed registers to be underestimated, the compiler may run out of scratch registers; it is essential that the allocator keep track of those registers that are free and busy, in order to detect such conditions.

Allocation of registers takes place as the result of a template match; the routine *allo* is called with a word describing the number of A registers, B registers, and temporary locations needed. The allocation of temporary locations on the stack is relatively straightforward, and will not be further covered; the bookkeeping is a bit tricky, but conceptually trivial, and requests for temporary space on the stack will never fail.

Register allocation is less straightforward. The two major complications are *pairing* and *sharing*. In many machines, some operations (such as multiplication and division), and/or some types (such as longs or double precision) require even/odd pairs of registers. Operations of the first type are exceptionally difficult to deal with in the compiler; in fact, their theoretical properties are rather bad as well.[9] The second issue is dealt with rather more successfully; a machine dependent function called *szty(t)* is called that returns 1 or 2, depending on the number of A registers required to hold an object of type *t*. If *szty* returns 2, an even/odd pair of A registers is allocated for each request.

The other issue, sharing, is more subtle, but important for good code quality. When registers are allocated, it is possible to reuse registers that hold address information, and use them to contain the values computed or accessed. For example, on the IBM 360, if register 2 has a pointer to an integer in it, we may load the integer into register 2 itself by saying:

```
L       2,0(2)
```

If register 2 had a byte pointer, however, the sequence for loading a character involves clearing the target register first, and then inserting the desired character:

```
SR      3,3
IC      3,0(2)
```

In the first case, if register 3 were used as the target, it would lead to a larger number of registers used for the expression than were required; the compiler would generate inefficient code. On the other hand, if register 2 were used as the target in the second case, the code would simply be wrong. In the first case, register 2 can be *shared* while in the second, it cannot.

In the specification of the register needs in the templates, it is possible to indicate whether required scratch registers may be shared with possible registers on the left or the right of the input tree. In order that a register be shared, it must be scratch, and it must be used only once, on the appropriate side of the tree being compiled.

The *allo* routine thus has a bit more to do than meets the eye; it calls *freereg* to obtain a free register for each A and B register request. *Freereg* makes multiple calls on the routine *usable* to decide if a given register can be used to satisfy a given need. *Usable* calls *shareit* if the register is busy, but might be shared. Finally, *shareit* calls *ushare* to decide if the desired register is actually in the appropriate subtree, and can be shared.

Just to add additional complexity, on some machines (such as the IBM 370) it is possible

to have "double indexing" forms of addressing; these are represented by OREGS's with the base and index registers encoded into the register field. While the register allocation and deallocation *per se* is not made more difficult by this phenomenon, the code itself is somewhat more complex.

Having allocated the registers and expanded the assembly language, it is time to reclaim the resources; the routine *reclaim* does this. Many operations produce more than one result. For example, many arithmetic operations may produce a value in a register, and also set the condition codes. Assignment operations may leave results both in a register and in memory. *Reclaim* is passed three parameters; the tree and cookie that were matched, and the rewriting field of the template. The rewriting field allows the specification of possible results; the tree is rewritten to reflect the results of the operation. If the tree was computed for side effects only (FOREFF), the tree is freed, and all resources in it reclaimed. If the tree was computed for condition codes, the resources are also freed, and the tree replaced by a special node type, FORCC. Otherwise, the value may be found in the left argument of the root, the right argument of the root, or one of the temporary resources allocated. In these cases, first the resources of the tree, and the newly allocated resources, are freed; then the resources needed by the result are made busy again. The final result must always match the shape of the input cookie; otherwise, the compiler error "cannot reclaim" is generated. There are some machine dependent ways of preferring results in registers or memory when there are multiple results matching multiple goals in the cookie.

The Machine Dependent Interface

The files *order.c*, *local2.c*, and *table.c*, as well as the header file *mac2defs*, represent the machine dependent portion of the second pass. The machine dependent portion can be roughly divided into two: the easy portion and the hard portion. The easy portion tells the compiler the names of the registers, and arranges that the compiler generate the proper assembler formats, opcode names, location counters, etc. The hard portion involves the Sethi–Ullman computation, the rewriting rules, and, to some extent, the templates. It is hard because there are no real algorithms that apply; most of this portion is based on heuristics. This section discusses the easy portion; the next several sections will discuss the hard portion.

If the compiler is adapted from a compiler for a machine of similar architecture, the easy part is indeed easy. In *mac2defs*, the register numbers are defined, as well as various parameters for the stack frame, and various macros that describe the machine architecture. If double indexing is to be permitted, for example, the symbol R2REGS is defined. Also, a number of macros that are involved in function call processing, especially for unusual function call mechanisms, are defined here.

In *local2.c*, a large number of simple functions are defined. These do things such as write out opcodes, register names, and address forms for the assembler. Part of the function call code is defined here; that is nontrivial to design, but typically rather straightforward to implement. Among the easy routines in *order.c* are routines for generating a created label, defining a label, and generating the arguments of a function call.

These routines tend to have a local effect, and depend on a fairly straightforward way on the target assembler and the design decisions already made about the compiler. Thus they will not be further treated here.

The Rewriting Rules

When a tree fails to match any template, it becomes a candidate for rewriting. Before the tree is rewritten, the machine dependent routine *nextcook* is called with the tree and the cookie; it suggests another cookie that might be a better candidate for the matching of the tree. If all else fails, the templates are searched with the cookie FORREW, to look for a rewriting rule. The rewriting rules are of two kinds; for most of the common operators, there are machine dependent rewriting rules that may be applied; these are handled by machine dependent functions that are called and given the tree to be computed. These routines may recursively call

order or *codgen* to cause certain subgoals to be achieved; if they actually call for some alteration of the tree, they return 1, and the code generation algorithm recanonicalizes and tries again. If these routines choose not to deal with the tree, the default rewriting rules are applied.

The assignment ops, when rewritten, call the routine *setasg*. This is assumed to rewrite the tree at least to the point where there are no side effects in the left hand side. If there is still no template match, a default rewriting is done that causes an expression such as

$$a \mathrel{+}= b$$

to be rewritten as

$$a = a + b$$

This is a useful default for certain mixtures of strange types (for example, when *a* is a bit field and *b* an character) that otherwise might need separate table entries.

Simple assignment, structure assignment, and all forms of calls are handled completely by the machine dependent routines. For historical reasons, the routines generating the calls return 1 on failure, 0 on success, unlike the other routines.

The machine dependent routine *setbin* handles binary operators; it too must do most of the job. In particular, when it returns 0, it must do so with the left hand side in a temporary register. The default rewriting rule in this case is to convert the binary operator into the associated assignment operator; since the left hand side is assumed to be a temporary register, this preserves the semantics and often allows a considerable saving in the template table.

The increment and decrement operators may be dealt with with the machine dependent routine *setincr*. If this routine chooses not to deal with the tree, the rewriting rule replaces

$$x\,{+}{+}$$

by

$$((x \mathrel{+}= 1) - 1)$$

which preserves the semantics. Once again, this is not too attractive for the most common cases, but can generate close to optimal code when the type of x is unusual.

Finally, the indirection (UNARY MUL) operator is also handled in a special way. The machine dependent routine *offstar* is extremely important for the efficient generation of code. *Offstar* is called with a tree that is the direct descendant of a UNARY MUL node; its job is to transform this tree so that the combination of UNARY MUL with the transformed tree becomes addressable. On most machines, *offstar* can simply compute the tree into an A or B register, depending on the architecture, and then *canon* will make the resulting tree into an OREG. On many machines, *offstar* can profitably choose to do less work than computing its entire argument into a register. For example, if the target machine supports OREGS with a constant offset from a register, and *offstar* is called with a tree of the form

$$expr + const$$

where *const* is a constant, then *offstar* need only compute *expr* into the appropriate form of register. On machines that support double indexing, *offstar* may have even more choice as to how to proceed. The proper tuning of *offstar*, which is not typically too difficult, should be one of the first tries at optimization attempted by the compiler writer.

The Sethi-Ullman Computation

The heart of the heuristics is the computation of the Sethi-Ullman numbers. This computation is closely linked with the rewriting rules and the templates. As mentioned before, the Sethi-Ullman numbers are expected to estimate the number of scratch registers needed to compute the subtrees without using any stores. However, the original theory does not apply to real machines. For one thing, the theory assumes that all registers are interchangeable. Real machines have general purpose, floating point, and index registers, register pairs, etc. The

theory also does not account for side effects; this rules out various forms of pathology that arise from assignment and assignment ops. Condition codes are also undreamed of. Finally, the influence of types, conversions, and the various addressability restrictions and extensions of real machines are also ignored.

Nevertheless, for a "useless" theory, the basic insight of Sethi and Ullman is amazingly useful in a real compiler. The notion that one should attempt to estimate the resource needs of trees before starting the code generation provides a natural means of splitting the code generation problem, and provides a bit of redundancy and self checking in the compiler. Moreover, if writing the Sethi-Ullman routines is hard, describing, writing, and debugging the alternative (routines that attempt to free up registers by stores into temporaries "on the fly") is even worse. Nevertheless, it should be clearly understood that these routines exist in a realm where there is no "right" way to write them; it is an art, the realm of heuristics, and, consequently, a major source of bugs in the compiler. Often, the early, crude versions of these routines give little trouble; only after the compiler is actually working and the code quality is being improved do serious problem have to be faced. Having a simple, regular machine architecture is worth quite a lot at this time.

The major problems arise from asymmetries in the registers: register pairs, having different kinds of registers, and the related problem of needing more than one register (frequently a pair) to store certain data types (such as longs or doubles). There appears to be no general way of treating this problem; solutions have to be fudged for each machine where the problem arises. On the Honeywell 66, for example, there are only two general purpose registers, so a need for a pair is the same as the need for two registers. On the IBM 370, the register pair (0,1) is used to do multiplications and divisions; registers 0 and 1 are not generally considered part of the scratch registers, and so do not require allocation explicitly. On the Interdata 8/32, after much consideration, the decision was made not to try to deal with the register pair issue; operations such as multiplication and division that required pairs were simply assumed to take all of the scratch registers. Several weeks of effort had failed to produce an algorithm that seemed to have much chance of running successfully without inordinate debugging effort. The difficulty of this issue should not be minimized; it represents one of the main intellectual efforts in porting the compiler. Nevertheless, this problem has been fudged with a degree of success on nearly a dozen machines, so the compiler writer should not abandon hope.

The Sethi-Ullman computations interact with the rest of the compiler in a number of rather subtle ways. As already discussed, the *store* routine uses the Sethi-Ullman numbers to decide which subtrees are too difficult to compute in registers, and must be stored. There are also subtle interactions between the rewriting routines and the Sethi-Ullman numbers. Suppose we have a tree such as

$$A - B$$

where A and B are expressions; suppose further that B takes two registers, and A one. It is possible to compute the full expression in two registers by first computing B, and then, using the scratch register used by B, but not containing the answer, compute A. The subtraction can then be done, computing the expression. (Note that this assumes a number of things, not the least of which are register-to-register subtraction operators and symmetric registers.) If the machine dependent routine *setbin*, however, is not prepared to recognize this case and compute the more difficult side of the expression first, the Sethi-Ullman number must be set to three. Thus, the Sethi-Ullman number for a tree should represent the code that the machine dependent routines are actually willing to generate.

The interaction can go the other way. If we take an expression such as

$$*(p + i)$$

where p is a pointer and i an integer, this can probably be done in one register on most machines. Thus, its Sethi-Ullman number would probably be set to one. If double indexing is possible in the machine, a possible way of computing the expression is to load both p and i into

registers, and then use double indexing. This would use two scratch registers; in such a case, it is possible that the scratch registers might be unobtainable, or might make some other part of the computation run out of registers. The usual solution is to cause *offstar* to ignore opportunities for double indexing that would tie up more scratch registers than the Sethi-Ullman number had reserved.

In summary, the Sethi-Ullman computation represents much of the craftsmanship and artistry in any application of the portable compiler. It is also a frequent source of bugs. Algorithms are available that will produce nearly optimal code for specialized machines, but unfortunately most existing machines are far removed from these ideals. The best way of proceeding in practice is to start with a compiler for a similar machine to the target, and proceed very carefully.

Register Allocation

After the Sethi-Ullman numbers are computed, *order* calls a routine, *rallo*, that does register allocation, if appropriate. This routine does relatively little, in general; this is especially true if the target machine is fairly regular. There are a few cases where it is assumed that the result of a computation takes place in a particular register; switch and function return are the two major places. The expression tree has a field, *rall*, that may be filled with a register number; this is taken to be a preferred register, and the first temporary register allocated by a template match will be this preferred one, if it is free. If not, no particular action is taken; this is just a heuristic. If no register preference is present, the field contains NOPREF. In some cases, the result must be placed in a given register, no matter what. The register number is placed in *rall*, and the mask MUSTDO is logically or'ed in with it. In this case, if the subtree is requested in a register, and comes back in a register other than the demanded one, it is moved by calling the routine *rmove*. If the target register for this move is busy, it is a compiler error.

Note that this mechanism is the only one that will ever cause a register-to-register move between scratch registers (unless such a move is buried in the depths of some template). This simplifies debugging. In some cases, there is a rather strange interaction between the register allocation and the Sethi-Ullman number; if there is an operator or situation requiring a particular register, the allocator and the Sethi-Ullman computation must conspire to ensure that the target register is not being used by some intermediate result of some far-removed computation. This is most easily done by making the special operation take all of the free registers, preventing any other partially-computed results from cluttering up the works.

Compiler Bugs

The portable compiler has an excellent record of generating correct code. The requirement for reasonable cooperation between the register allocation, Sethi-Ullman computation, rewriting rules, and templates builds quite a bit of redundancy into the compiling process. The effect of this is that, in a surprisingly short time, the compiler will start generating correct code for those programs that it can compile. The hard part of the job then becomes finding and eliminating those situations where the compiler refuses to compile a program because it knows it cannot do it right. For example, a template may simply be missing; this may either give a compiler error of the form "no match for op ..." , or cause the compiler to go into an infinite loop applying various rewriting rules. The compiler has a variable, *nrecur*, that is set to 0 at the beginning of an expressions, and incremented at key spots in the compilation process; if this parameter gets too large, the compiler decides that it is in a loop, and aborts. Loops are also characteristic of botches in the machine-dependent rewriting rules. Bad Sethi-Ullman computations usually cause the scratch registers to run out; this often means that the Sethi-Ullman number was underestimated, so *store* did not store something it should have; alternatively, it can mean that the rewriting rules were not smart enough to find the sequence that *sucomp* assumed would be used.

The best approach when a compiler error is detected involves several stages. First, try to get a small example program that steps on the bug. Second, turn on various debugging flags in

the code generator, and follow the tree through the process of being matched and rewritten. Some flags of interest are −e, which prints the expression tree, −r, which gives information about the allocation of registers, −a, which gives information about the performance of *rallo*, and −o, which gives information about the behavior of *order*. This technique should allow most bugs to be found relatively quickly.

Unfortunately, finding the bug is usually not enough; it must also be fixed! The difficulty arises because a fix to the particular bug of interest tends to break other code that already works. Regression tests, tests that compare the performance of a new compiler against the performance of an older one, are very valuable in preventing major catastrophes.

Summary and Conclusion

The portable compiler has been a useful tool for providing C capability on a large number of diverse machines, and for testing a number of theoretical constructs in a practical setting. It has many blemishes, both in style and functionality. It has been applied to many more machines than first anticipated, of a much wider range than originally dreamed of. Its use has also spread much faster than expected, leaving parts of the compiler still somewhat raw in shape.

On the theoretical side, there is some hope that the skeleton of the *sucomp* routine could be generated for many machines directly from the templates; this would give a considerable boost to the portability and correctness of the compiler, but might affect tunability and code quality. There is also room for more optimization, both within *optim* and in the form of a portable "peephole" optimizer.

On the practical, development side, the compiler could probably be sped up and made smaller without doing too much violence to its basic structure. Parts of the compiler deserve to be rewritten; the initialization code, register allocation, and parser are prime candidates. It might be that doing some or all of the parsing with a recursive descent parser might save enough space and time to be worthwhile; it would certainly ease the problem of moving the compiler to an environment where *Yacc* is not already present.

Finally, I would like to thank the many people who have sympathetically, and even enthusiastically, helped me grapple with what has been a frustrating program to write, test, and install. D. M. Ritchie and E. N. Pinson provided needed early encouragement and philosophical guidance; M. E. Lesk, R. Muha, T. G. Peterson, G. Riddle, L. Rosler, R. W. Mitze, B. R. Rowland, S. I. Feldman, and T. B. London have all contributed ideas, gripes, and all, at one time or another, climbed "into the pits" with me to help debug. Without their help this effort would have not been possible; with it, it was often kind of fun.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, "Lint, a C Program Checker," Comp. Sci. Tech. Rep. No. 65 (1978).
3. A. Snyder, *A Portable Compiler for the Language C,* Master's Thesis, M.I.T., Cambridge, Mass. (1974).
4. S. C. Johnson, "A Portable Compiler: Theory and Practice," *Proc. 5th ACM Symp. on Principles of Programming Languages,* pp. 97-104 (January 1978).
5. M. E. Lesk, S. C. Johnson, and D. M. Ritchie, *The C Language Calling Sequence,* Bell Laboratories internal memorandum (1977).
6. S. C. Johnson, "Yacc — Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
7. A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *J. Assoc. Comp. Mach.* 23(3) pp. 488-501 (1975). Also in *Proc. ACM Symp. on Theory of Computing,* pp. 207-217, 1975.
8. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. Assoc. Comp. Mach.* 17(4) pp. 715-728 (October 1970). Reprinted as pp. 229-247 in *Compiler Techniques,* ed. B. W. Pollack, Auerbach, Princeton NJ (1972).
9. A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," *Proc. 4th ACM Symp. on Principles of Programming Languages,* pp. 21-28 (January 1977).

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

April 3, 1978

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

---

†UNIX is a Trademark of Bell Laboratories.

PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time; it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search*. Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time;

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.

CONCLUSIONS

On the issue of password security, UNIX is probably better than most systems. The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

It is also worth some effort to conceal even the encrypted passwords. Some UNIX systems have instituted what is called an "external security code" that must be typed when dialing into the system, but before logging in. If this code is changed periodically, then someone with an old password will likely be prevented from using it.

Whenever any security procedure is instituted that attempts to deny access to unauthorized persons, it is wise to keep a record of both successful and unsuccessful attempts to get at the secured resource. Just as an out-of-hours visitor to a computer center normally must not only identify himself, but a record is usually also kept of his entry. Just so, it is a wise precaution to make and keep a record of all attempts to log into a remote-access time-sharing system, and certainly all unsuccessful attempts.

Bad guys fall on a spectrum whose one end is someone with ordinary access to a system and whose goal is to find out a particular password (usually that of the super-user) and, at the other end, someone who wishes to collect as much password information as possible from as many systems as possible. Most of the work reported here serves to frustrate the latter type; our experience indicates that the former type of bad guy never was very successful.

We recognize that a time-sharing system must operate in a hostile environment. We did not attempt to hide the security aspects of the operating system, thereby playing the customary make-believe game in which weaknesses of the system are not discussed no matter how apparent. Rather we advertised the password algorithm and invited attack in the belief that this approach would minimize future trouble. The approach has been successful.

References

[1] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Comm. ACM* **17** (July 1974), pp. 365-375.

[2] *Proposed Federal Information Processing Data Encryption Standard.* Federal Register (40FR12134), March 17, 1975

[3] Wilkes, M. V. *Time-Sharing Computer Systems.* American Elsevier, New York, (1968).

[4] U. S. Patent Number 2,089,603.

On the Security of UNIX

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known— but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
        mkdir x
        cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems (almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of

†UNIX is a Trademark of Bell Laboratories.

the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect *and*-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login*, allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited.

The encryption scheme used is not one of the strongest known, but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su*. A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and t pes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.