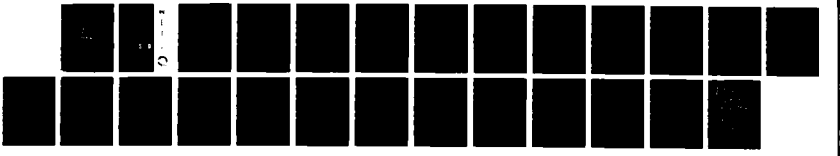
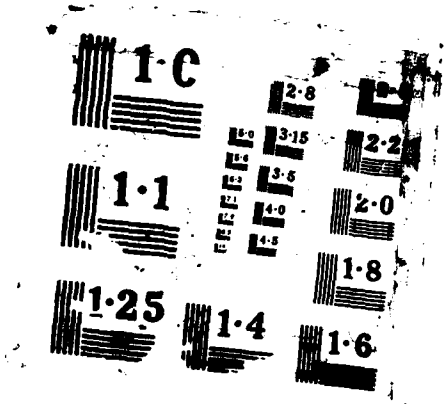
AD-A193 375 A SIMPLE COMPUTER DATA BASE SYSTEM FOR UNIX(U) ARMY ENGINEER TOPOGRAPHIC LABS FORT BELVOIR VA M M MCDONNELL MAR 88 ETL-0494 1/1

UNCLASSIFIED F/G 12/5 NL




1·C
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·8
3·15
3·6
4·0
4·5

(4)

ETL-0494

A simple computer data base system for UNIX

Michael M. McDonnell

March 1988

U.S. ARMY CORPS OF ENGINEERS
ENGINEER TOPOGRAPHIC LABORATORIES
FORT BELVOIR, VIRGINIA 22060-5546

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188
Exp. Date: Jun 30, 1986

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) ETL-0494	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION U.S. Army Engineer Topographic Laboratories 6b. OFFICE SYMBOL (If applicable) CEETL-RI-I	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Fort Belvoir, Virginia 22060-5546	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. 4A161102B52C / TASK NO. C / WORK UNIT ACCESSION NO 14

11. TITLE (Include Security Classification)
A Simple Computer Database System for UNIX

12 PERSONAL AUTHOR(S)
McDonnell, Michael M.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 1985 TO 1986	March 1988	19

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	UNIX, String, database, inventory, rolodex, computer program.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This ~~is a~~ computer program that allows users to maintain and access a file containing addresses, inventory items, or other units of text information grouped in blocks separated by blank lines. Any string within a file may be used to find and print the block(s) of test containing the string. A file is created, maintained, and accessed by a group of UNIX programs which have been designed for speed and simplicity. Besides being useful in themselves, these programs illustrate cooperative use of C programs and shell command files. A history of the development will also be given since this is of general interest to programmers.

This program uses standard UNIX techniques, except for the Boyer-Moore string matching algorithm. It offers a simple and extensible approach to the type of database represented by the rolodex file found in many offices. This simple flat-file database has proven valuable as a way of maintaining and accessing an inventory file and an address file.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Michael McDonnell	22b TELEPHONE (Include Area Code) 202/355-2724 / 22c. OFFICE SYMBOL CEETL-RI-I

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block #19 (continued)

The data file is a plain text file containing no control characters aside from new lines. The file is therefore easy to create and maintain using ordinary text editors, though a program is provided to facilitate item entry for users. On an unloaded VAX 780 it takes about 1.5 seconds to search a data file of 150,000 characters. On a system which is about 10 users, this time is about 3 seconds.

PREFACE

This study was conducted under DA Project 4A161102B52C, Task C, Work Unit 14, "Artificial Intelligence Research."

The study was done during the period 1985 - 1986 under the supervision of Ms. Anne Werkheiser, Team Leader, Center for Artificial Intelligence, and Mr. Lawrence A. Gambino, Director, Research Institute.

COL Alan L. Laubscher, EN, was Commander and Director, and Mr. Walt Boge was Technical Director of the Engineer Topographic Laboratories during the study period.

TABLE OF CONTENTS

ILLUSTRATIONS

A SIMPLE COMPUTER DATABASE SYSTEM FOR UNIX*

INTRODUCTION

This report is about a computer program developed at the U.S. Army Engineer Topographic Laboratories. The program, named Boyer-Moore Blockprinter [**bmb**], is a utility, or useful program, that allows users to maintain and access a file containing addresses, inventory items, or other units of text information grouped in blocks separated by blank lines. Any string within a file may be used to find and print the block(s) of text containing the string. A file is created, maintained, and accessed by users through a group of Unix* programs which have been designed for speed and simplicity. Besides being useful in themselves, these programs illustrate cooperative use of C programs and shell command files. Testing reveals that the search program is approximately 15 times faster than a similar program using simple string-search algorithms. This report will show and discuss the code used and will provide a history of the development since this is of general interest to programmers.

This utility uses standard Unix techniques, though the Boyer-Moore string matching algorithm is not yet a Unix standard. It offers a simple and extensible approach to the type of database represented by the Rolodex** file found in many offices. This simple flat-file database has proven valuable at ETL as a way of maintaining and accessing an inventory file and, of course, an address file. It has proven useful and is presently used by several people on a daily basis.

The data file is a plain text file containing no control characters aside from newlines [ctrl-J]. The file is therefore easy to create and maintain using ordinary text editors (word processors), although a program is provided to facilitate item entry for users.

On an unloaded VAX 780 it takes about 1.5 seconds to search a data file of 150,000 characters. On a system with about 10 users, this time is about 3 seconds. Detailed timings are given in appendix B.

First, the background of the problem and a history of its solution will be presented. Then a short tutorial on Boyer-Moore string matching will be provided, and finally the general applicability of this technique will be discussed. The appendixes have complete source code listings in C and Bourne shell scripts for a Unix system as well as some test results.

---

* Unix is a registered trademark of AT&T Technologies.

** Rolodex is a registered trademark of the Rolodex Corporation.

BACKGROUND

Owing to problems accessing inventory items stored in a computer file, a program was written to pull out items by searching for any string, a sequence of characters, in the file. The program was quite slow, taking about 45 seconds for each query, but it was welcomed by its users. This justified an effort to speed it up.

The program was well received for several reasons. Under the previous inventory record system, there were multiple copies of item records kept by different people. At least partly owing to this separate record-keeping, there was uncertainty about who had the current record of an item and who was responsible for seeing that the records were correct. Formats for entering a new item were also not standardized. Because of this, there was considerable trouble finding equipment, especially each year at inventory time. A first step was made by our administrative assistant who had the inventory typed into the VAX in a standard format, but no good method was available for searching out specific items. The solution adopted was to print out the entire inventory and work from the listing. Because this was unsatisfactory, a program was written to allow people to search only for those items that they needed at the time.

Initial methods used to speed up the program were C programming techniques such as using register variables and using a large buffer to read the file to be searched into the computer. Although These techniques speeded up a typical search to about 30 seconds per query, this was still not good enough. The solution seemed to be a superior string-searching technique, the Boyer-Moore method, which had been discussed in several places, most notably in a Scientific American article on programming[1].

A literature search discovered several articles about Boyer-Moore string searching, but most did not give an algorithm detailed enough to program. De V. Smit's article[2], while not correct in every point, did give enough information to finish the job. The result was an astonishing speedup by a factor of 10, bringing search time to about 3 seconds.

The current version of the program uses an insight from Dan Hoey[3] which is that conversion of all letters in text to lower case is not essential if all letters are not searched, and indeed they are not in Boyer-Moore pattern matching. Incorporation of this suggestion speeded up the program by about another factor of two so that a search that originally took 45 seconds is now finished in one and

---

[1] N. Wirth. *Programming Languages*, Scientific American, September 1984.

[2] G. De V. Smit. *A Comparison of Three String Matching Algorithms*, Software - Practice and Experience, Vol. 12, 57-66, 1982.

[3] Personal Communication. Mr. Hoey works at the Navy Center for Applied Research in Artificial Intelligence, Washington, DC.

one-half seconds, a speedup of 30 times, most of which is owing to using the Boyer-Moore technique.

BOYER-MOORE STRING MATCHING

To define our terms, call the sequence of characters that the user wants to look for the "string" and call the text file to be searched the "text". The simplest way to search for a string in the text is to start at the beginning of both the string and the text and compare letters from front to back. If all the letters match up to the end of the string, we have a "match" of the string against the text. If a mismatch is found before reaching the end of the string, then the string is moved forward one space and matching starts again from the front of the string and from the second character in the text. This process is repeated until you find a match or until you reach the end of the string, in which case the match fails. Figure 1 shows this process.

```
                                        COUNT
THIS IS A TEXT STRING
TEXT                                      2

THIS IS A TEXT STRING
 TEXT                                     1

THIS IS A TEXT STRING
  TEXT                                    1

THIS IS A TEXT STRING
   TEXT                                   1

THIS IS A TEXT STRING
    TEXT                                  1

THIS IS A TEXT STRING
     TEXT                                 1

THIS IS A TEXT STRING
      TEXT                                1

THIS IS A TEXT STRING
       TEXT                               1

THIS IS A TEXT STRING
        TEXT                              1

THIS IS A TEXT STRING
         TEXT                             1

THIS IS A TEXT STRING
          TEXT                            4
                                        -------
           TOTAL NUMBER OF TRIES          15
```

Figure 1. The straightforward algorithm.

In Figures 1 and 2, the caret under the string shows what letters have been checked at each step. These figures should be seen as top-to-bottom movies showing how the algorithms proceed. A count of character checks is kept on the right side and totaled at the bottom.

The method just described, which De V. Smit calls the StraightForward (SF) algorithm, is quite slow as can be seen from the fact that a given character in the text may be checked several times. An improvement may be seen in the case where the mismatched letter in the text does not appear in the string at all. In this case, you can move the string all the way past the mismatched letter and continue matching from there as is shown in Figure 2. An elaboration of this idea is found in the Knuth-Morris-Pratt (KMP)[4] algorithm which guarantees that a given letter in the text will be checked exactly once. This is shown to be the best that you can do if you search from the front of both the string and the text as in the SF method.

Boyer and Moore[5] had the insight that if you start matching from the *back* of the string rather than from the front, then there will typically be letters in the text that you don't need to examine at all. Figure 2 shows how this works for our test case. Here the number of tries is only half what it was in the SF case, but there is even greater savings than this since the string only has to be moved against the text 3 times here while it had to be moved 10 times in the SF case. The KMP method is not shown, but it would have required 14 tries since there are 14 characters between the beginning of the text and the match point. In tests against a large amount of text, De V. Smit could find no advantage to the KMP method over the SF method since KMP requires a fairly large overhead setting up the index tables for jumping the string over the text, and it does not provide a significant decrease in the number of tries required.

```
                                        COUNT
THIS IS A TEXT STRING
TEXT                                      1

THIS IS A TEXT STRING
    TEXT                                  1

THIS IS A TEXT STRING
       TEXT                               1

THIS IS A TEXT STRING
         TEXT                             4
                                        -------
       TOTAL NUMBER OF TRIES              7
```

Figure 2: Boyer-Moore string matching

[4] D. Knuth, J. Harris Jr., V. Pratt. *Fast Pattern Matching In Strings*, SIAM J. Computing, Vol. 6, 323-350, 1977.

[5] R. Boyer, J. Moore. *A Fast String Search Algorithm*, Comm. ACM, Vol. 20, 762-772, 1977.

Counting the first text letter as number 1, the Boyer-Moore (BM) algorithm does not even check letters 1, 2, 3, 5, 6, 7, 9, or 10 in the text. The proportion of unchecked letters grows even larger for longer strings, which can make bigger jumps against the text.

There is a price to be paid for BM string matching. Two jump tables have to be initialized before the matching starts (see comments on the C program for the details of building these jump tables). Despite this, De V. Smit's randomized tests (using Afrikaans text!) showed that

> When patterns longer than 3 characters are searched, when the expected penetration is more than 100 characters and when backing up in the text string will not pose problems, the BM algorithm is by far the best algorithm to use.[6]

Our requirements, and most string search problems, easily meet these criteria.

OTHER POSSIBLE APPLICATIONS

The suite of programs presented in appendix A can be modified by users to create and maintain their own databases of personal property items, phone numbers, parts catalogs, suppliers of services, and whatever other small databases *may serve their needs.*

It is possible to do this since the **bmb** program is not "privileged" and can be used by anybody in the form of a Unix shell command personalized to their needs. As an example, a user named Mary can establish an on-line "little black book" of phone numbers and make it private by giving it an access code, on Unix, of 600. This means that nobody else can read or write this file, only Mary. Now if she calls the file "bbook" and puts it in her home directory as, for example, /usr/mary/bbook, then she can write a shell command like this:

```
/local/bin/bmb $* /usr/mary/bbook
```

Which she can call, say, mbb and put in her execution search path. She can then look up all the Mikes in her book by a command like

```
mbb mike
```

and the information blocks for all the Mikes will be printed. If anyone else tries to execute her program, they will get a message telling them that they don't have access permission to her bbook file. So we have a public program allowing private access, the best of both worlds. Notice that case translation is done so that asking for "mike" will also find "Mike", "MIKE", or even "MiKey".

---

[6] G. De V. Smit. *A Comparison of Three String Matching Algorithms*, Software - Practice and Experience, Vol. 12, 57-66, 1982.

Another advantage of this technique over other database methods is the lack of keywords or other restricted forms of reference. This is useful because you may not know in advance what strings you want to use to retrieve information in the future. You are absolved of the necessity of deriving such information, and of typing it in. The format of an entry is also very free-form. The only restriction is a blank line required between blocks of text. This prevents the problem, prevalent in many databases, of forcing your entry into a restricted format or, conversely, of wasting a large block of disk storage for a one-line entry.

Bmb can have an important impact in the workplace. For example, the **rolo** program, based on **bmb**, started out as an address file but, through use, it evolved into an administrative database. We found that **rolo** could automate the "getting and presenting information" performance element that is so often required. For example we might want our getter-and-presenter-of-information to find out who to talk to in order to get another consulting contract with Dr. Hacker. If done manually, this task requires a search through hard-copy files to find out how it was done last time and who was talked to then. But the command **rolo hacker** will print the answer set:

> Dr. Manly Hacker
> University of California at San Diego
> (619) 555-2424
>
> Mr. Frederick Papershuffler
> Department of Organization
> 555-8986
> (he worked on Dr. Hacker's contract, see /admin/hacker/contract)

Thus, a frequently-performed function has been automated.

Using **rolo** in this way can automate the job of any getters-and-presenters. For example, if a user wanted to consult with a topologist he would normally have to find a mathematician and then ask him for a list of them. But **rolo topolo** produces:

> Dr. Samuel Manifold
> Coast Guard Laboratories
> 555-8734
> (he's a topologist)
>
> Joseph Sequence ext 259
> Joe knows a lot about topology considering he's a programmer.

which is the required information.

Using **rolo** as an administrative tool instead of just an address book is the result of an organic process of trial and experiment. The most useful features of **rolo** are those that flexibly allow the user to design his own uses. Nothing is

designed in, there are no fixed formats, every user can use it differently. Our experience at ETL is that each year we have fewer pieces of paper to handle and are less often at a loss for information. This program has, therefore, proven to be a valuable means of organizing information and serves adequately as a database of corporate memory.

The U.S. Government owns this program, but you are free to copy it for your own use and to distribute it to other people. Further, you may freely modify it to suit your specific requirements. This has proven to be one of the most important characteristics of the program, that it can be easily modified to suit specific situations. Please feel free to use **bmb** in your work. It is much quicker to "**rolo**" a name than it is to look it up, even in a real Rolodex^{tm}. Appendix A is a full source code listing for **bmb** and **rolo**. Appendix B includes tests of the efficiency of **bmb** and compares it to the SF method as well as to earlier versions of itself.

APPENDIX A -- PROGRAM SOURCE CODE

Here are listings of the C program **bmb** and shell scripts used in the **bmb** variants **rolo** and **inveno**.

```
/*
 * bmb.c
 *
 * Boyer-Moore Case-independent, Block printing.
 *
 * This program uses the text searching algorithm of Boyer and Moore as
 * reported by G. De V. Smit in Software - Practice and Experience
 * vol. 12, pp. 56-66 (1982). also see: CACM Sept '79 and Oct '77
 *
 * The search is case-independent.
 *
 * The method allows fairly large jumps to be taken in comparing the
 * "string" against the "text".  Whitespace is allowed in "string",
 * so a whole phrase or sentence or more can be searched for.
 *
 * A table is built starting at the back end of the string (location
 * zero) and counting backwards. The position of each letter from the
 * end of the string is entered in the table.  Only the first
 * occurrence from the back is entered for each letter.  If a letter
 * does not appear in the string, it is given a value of the whole
 * length of the string. The last letter in the string gets a value
 * of zero.
 *
 * A second table is also built indicating similar substrings in the
 * string. This table is used to offset the string until a substring
 * closer to the front of the string matches the indicated substring
 * in text.  If no similar substrings are present, the offset is
 * equal to the length of the string plus the distance from the back
 * of the string that has been searched so far (i.e. to the point of
 * a mismatch).
 *
 * The string is checked for a match against text by starting from the
 * back of the string and matching against the front of the text.
 * When a mismatch is found, the mismatched letter in text is used to
 * index into the first offset table, and the mismatched letter in
 * the string is used to index into the second offset table.  The
 * string is moved a distance equal to the maximum of the two offsets
 * derived from the tables, and the letter-by-letter check starts
 * again from the back. This method allows more or less large jumps
 * to be made along the text.  This means that the text should be
 * buffered.  To minimize disk access, a large text buffer is used.
 *
 * -Mike McDonnell  86/01/06
 *   U. S. Army Engineer Topographic Laboratories (ETL)
 *   Bldg. 2592 Ft. Belvoir, VA  22060-5540
 * tel:(202)355-2724 arpanet: mike@etl
 */
```

```
#include        <stdio.h>
#include        <sys/file.h>
#define TBUF (BUFSIZ * 8)
#define ISUPPER(c) ((c)>='A' && (c)<='Z')
#define TOLOWER(c) ((c)+('a'-'A'))
#define LOWER(c) (ISUPPER(c) ? TOLOWER(c) : (c))
#define MIN(a,b) (((a) < (b)) ? (a) : (b))
#define MAX(a,b) (((a) > (b)) ? (a) : (b))


int     offset1[128],           /* enough room for all ASCII chars */
        offset2[128];
char    textbuf[TBUF];          /* the text */

main(argc, argv)
  int     argc;
  char   *argv[];

{
  register char *sp,            /* moving pointer to string */
         *tp,                   /* pointer to text buffer */
         *strlast,              /* last letter in string */
         *strfirst,             /* first letter in string */
         *textlast;             /* pointer to end of text */
  register int i, t;
  int     slen;                 /* length of string */
  int     fd;                   /* file descriptor for text file */
  char   *match();
  int     temp[128];

  if (argc < 3)
  {
    printf("\nusage: %s string textfile\n", argv[0]);
    exit(1);
  }
  if ((fd = open(argv[2], O_RDONLY)) < 0)
    perror("can't open text file"), exit(1);

  strfirst = sp = argv[1];
  for (slen = 0; *sp != '\0'; ++sp, ++slen)
    *sp = LOWER(*sp);
  strlast = --sp;

  /* init all offset1[] to string length */
  for (i = 0; i <= 127; ++i)
    offset1[i] = slen;

  /* load offset1[], init offset2[] */
  for (i = slen, sp = strfirst; i >= 1; --i)
  {
    if (offset1[(int) *(sp + i - 1)] == slen)
      offset1[(int) *(sp + i - 1)] = slen - i;
    offset2[i] = 2 * slen - i;
  }
```

```
  /* load offset2[].  This is hard to understand. */
  for (i = slen, t = slen + 1; i > 0; --i, --t)
    for (temp[i] = t; t <= slen && sp[i - 1] != sp[t - 1]; t = temp[t])
      offset2[t] = MIN(offset2[t], slen - i);
  for (i = 1; i <= t; ++i)
    offset2[i] = MIN(offset2[i], slen + t - i);
  for (i = temp[t]; t <= slen; i = temp[i])
    for (; t <= i; ++t)
      offset2[t] = MIN(offset2[t], i - t + slen);


  /* #define DEBUG                          /*  */
#ifdef DEBUG
  for (sp = strfirst, i = 1; sp <= strlast; ++sp, ++i)
    printf("offset1[%c] = %d   offset2[%d] = %d\n",
           *sp, offset1[(int) *sp], i, offset2[i]);
#endif
#undef DEBUG

  /* The offset table is now full.  It's time to start searching */
  while ((i = read(fd, textbuf, TBUF)) > 0)
  {

    /*
     * Don't split a block; set the file pointer back to a double
     * newline
     */
    textlast = textbuf + i - 1;
    if (i == TBUF)                /* only if we have a full buffer */
    {
      for (tp = textlast; *tp != '\n' || *(tp + 1) != '\n'; --tp)
        ;
      textlast = tp + 1;
      lseek(fd, (long) (tp - textbuf - i), 1);
    }
    tp = textbuf + slen - 1;

    /* Do it!  Go baby, go! */
    for (sp = strlast;; sp = strlast)
    {
      for (; *sp == LOWER(*tp) && sp != strfirst; --sp, --tp)
        ;
      /* if mismatch, step ahead */
      if (*sp != LOWER(*tp))
      {
        tp += MAX(offset1[LOWER(*tp)], offset2[(sp - strfirst) + 1]);
        if (tp > textlast)
          break;                  /* all done with buffer */
      }
      else                        /* match! */
        tp = match(tp);
    }
  }
  putchar('\n');
}
```

```
/* print the block in which a match occurred */
char   *
match(ptext)
  char   *ptext;
{
  register char *cp;

  cp = ptext;
  /* back up to beginning of block (a blank line) */
  while (*cp != '\n' || *(cp - 1) != '\n')
    --cp;
  while (*cp != '\n' || *(cp + 1) != '\n')
    putchar(*cp++);
  putchar('\n');
  return (cp);
}
```

Shell scripts to run **bmb** on particular files can be written like the following. Each line is a separate program. These two are called "**rolo**" and "**inveno**" respectively:

```
/local/bin/bmb $* /local/rolodex

/local/bin/bmb $* /local/inventory
```

Next is a shell script which checks to see if the rolodex file has grown and backs it up in a safe place if it has. This gives an extra level of protection of the rolodex file since anyone can edit the file. This program has not proven necessary in practice since nobody has damaged the rolodex file, whether accidentally or maliciously, since the file was started over four years ago. As a general matter of system philosophy, we leave the system as wide-open to users as possible. We did have some initial misgivings about this, but it has proved to be the correct choice since people learn more about a system they can "play" with, and the system administrator's burden is lessened since people can correct bugs themselves. In practice, naive users do not get into things they don't know about very often.

```
#! /bin/sh
#
#       roloback
#
# This program performs periodic backup of the on-line electronic
# rolodex file.  It is triggered by an entry in crontab.
# The rolodex file is only backed up if it has grown larger than the
# backup file.  This is a good heuristic, since if someone wrecks a
# file with an editor they usually won't make it larger.
#                       -Mike McDonnell  84/01/18
#
ROLO=/usr/local/rolodex
BACK=/cai/mike/rolo/rolodex.backup

rolosize=`ls -l $ROLO | awk '{ printf("%d", $4) }'`
backupsize=`ls -l $BACK | awk '{ printf("%d", $4) }'`

if [ $rolosize -gt $backupsize ]
then
        echo "Backing up the rolodex file..."          >/dev/console
        cp $ROLO $BACK
fi
```

Next is script "**rolin**" which allows people to more-or-less easily make new entries to the rolodex file. Of course, if you make a mistake, it must be corrected by using a text editor on the rolodex file, but this usually works fine. A nicety not mentioned here is that any file used by **bmb** must *begin* with a blank line as well as ending with one. These blank lines act as sentinels to prevent the text pointers from straying outside the text buffer. Alert readers will notice that the "24 lines per message" warning is not enforced by the code. It is there to prevent users from being too wordy.

```
#! /bin/sh
#
#       rolin
#
# This is a program to append a new message onto file
# /usr/local/rolodex, our electronic address book.
#               -mmm  84/01/07
ROLODEX=/usr/local/rolodex
echo "Enter address and/or comments. No blank lines are allowed."
echo "Maxes are 80 characters per line and 24 lines per message."
echo "Press ctl-D when done."
echo ""
cat >>$ROLODEX

# add a blank line to the end of the message
echo "" >>$ROLODEX
```

APPENDIX B -- TESTS

This is a file showing how fast **bmb** works in searching some text files. File "rolodex" is an address, telephone number and miscellaneous information file. File "inventory" is our entire accountable property file. Comments have been made throughout this file explaining what is being tested. Here is the UNIX shell script used to generate the test results. As you can see, it is a useful little program by itself and can be used for timing any function.

Here are the sizes of the test text files from the UNIX utility **wc**:

```
  820   3250  23792     /local/rolodex
 9372  17333 144245     /local/inventory
```

```
#! /bin/sh
# test execution time of a program
#
ITER=25
count=0
trap 'rm -f /tmp/cruft$$; exit 1' 1 2 15
echo "Testing: $ITER iterations of \"$1\""
while [ $count -lt $ITER ]
do
        /bin/time $1 2>>/tmp/cruft$$ >/dev/null
        count=`expr $count + 1`
done
cat /tmp/cruft$$ | awk 'BEGIN { wall = 0; user = 0; system = 0; count = 0 }
            { wall += $1; user += $3; system += $5; count += 1 }
        END { printf("\nWall clock average   = %f", wall/count)
              printf("\nUser time average    = %f", user/count)
              printf("\nSystem time average  = %f\n\n\n", system/count) }'
rm /tmp/cruft$$
```

TEST RESULTS

Effect of Buffer Size

Some tests were run using different size buffers in the program to minimize disk access. The optimum for a VAX780 seems to be about 4 to 8 kilobytes (KB). The final version uses an 8 KB buffer. The buffer size is given before the start of each test run. These tests were done with four users on the system and a load average of about 2.0. The string "computer" gets about 25 hits.

```
Buffer size is 32 KB
Testing: 25 iterations of "./bigbmb computer /local/inventory"
Wall clock average   = 1.692000
User time average    = 0.688000
System time average  = 0.340000

Buffer size is 8 KB
Testing: 25 iterations of "./bmb computer /local/inventory"
Wall clock average   = 1.688000
User time average    = 0.696000
System time average  = 0.340000

Buffer size is 2 KB
Testing: 25 iterations of "./littlebmb computer /local/inventory"
Wall clock average   = 1.820000
User time average    = 0.696000
System time average  = 0.476000
```

Comparison With Simple String Search

Here are some results from the grep family for comparison. Grep uses a simple front-to-back string search. Note that **bmb** is doing case translation and is backing up a pointer and printing out a block of text (typically 6 lines) for each hit, so its performance is even more impressive than it seems at first.

```
Testing: 25 iterations of "egrep computer /local/inventory"
Wall clock average   = 5.392000
User time average    = 1.872000
System time average  = 0.788000

Testing: 25 iterations of "fgrep computer /local/inventory"
Wall clock average   = 6.664000
User time average    = 3.348001
System time average  = 0.576000

Testing: 25 iterations of "grep computer /local/inventory"
Wall clock average   = 5.440001
User time average    = 2.380000
System time average  = 0.612000
```

Incremental Improvement

These results show how various versions of **bmb** got faster. Techniques used include register pointers and a large text buffer, but the biggest gain in speed came from using the Boyer-Moore algorithm. Program "**inveno**" is like **bmb** with the pathname of the inventory file hard-coded in.

```
oldrolo/inveno has the pre-Boyer-Moore stuff, quite slow.
Testing: 25 iterations of "./oldrolo/inveno computer"
Wall clock average   = 27.972004
User time average    = 14.180001
System time average  = 0.628000

This was a first cut at using Boyer-Moore.  The stuff in oldrolo2 uses a
separate lower-case buffer, which was discarded after I heard Dan Hoey's
(hoey@nrl-aic) suggestion.
Testing: 25 iterations of "./oldrolo2/bmb computer /local/inventory"
Wall clock average   = 2.852000
User time average    = 1.576000
System time average  = 0.376000

The current version:
Testing: 25 iterations of "./bmb computer /local/inventory"
Wall clock average   = 1.536000
User time average    = 0.700000
System time average  = 0.336000
```

Shell Script Versus Hard-Coded Pathname

This shows the difference in speed between running **bmb** as a shell script versus having the file pathname hard-coded in. There was not much difference, and the general-purpose **bmb** is much more useful than a fixed-path program since it allows users to keep their own data files without having their own version of **bmb**.

```
Testing: 25 iterations of "./inveno computer"
Wall clock average   = 1.628000
User time average    = 0.688000
System time average  = 0.336000

Testing: 25 iterations of "./script_inveno computer"
Wall clock average   = 1.824000
User time average    = 0.724000
System time average  = 0.480000
```

Test On A Large File

Finally, a test of **bmb** on a fairly large file. /local/bmbtestfile is a 1Mbyte file of concatenated inventory files. its wc is:

65604 121331 1009715

and there are 252 hits on the string "telescope" since wc of "egrep TELESCOPE /local/bmbtestfile" is

35 252 1967

This is probably about as large a file as you would want to use **bmb** on. Conventional databasing techniques become more efficient beyond this file size, but this is a big enough file to be useful for many tasks.

```
Testing: 25 iterations of "bmb telescope /local/bmbtestfile"
Wall clock average   = 8.576000
User time average    = 4.356000
System time average  = 2.064000

Testing: 25 iterations of "egrep TELESCOPE /local/bmbtestfile"
Wall clock average   = 18.515999
User time average    = 12.919999
System time average  = 4.812000
```

END

DATE

FILMED

DTIC

JULY 88