DI-AD-EME

Betty Christensen

July 24, 1985

Dr. Roy Ellzey _________________________ Chrm.

Dr. David Thomas _______________________

Robert Diersing _________________________
ABSTRACT

It is the purpose of this project to significantly improve the input and output performance of the VAX 11/750 by providing a direct address method for the UNIX data files. The programs of DI-AD-EME create a user-friendly environment which encourages more accurate input from the user while increasing the speed of real time processing. This has been accomplished by standardizing the files to fixed length records, building sequentially ordered indices, and providing the capability of searching the indices using a binary search. The physical order of the indices may be checked a separate program whenever verification is needed. The advantages gained by using DI-AD-EME techniques can be shared with any program on the system simply by calling the access program, which returns the physical offset address of the desired record to the calling program.
TABLE OF CONTENTS

I. Introduction

II. Programs
   A. menu.c
   B. create.c
   C. add.c
   D. update.c
   E. delete.c
   F. access.c
   G. DIcheck.c

III. Summary and Conclusions

IV. Bibliography

V. Appendix
   A. menu.c
   B. create.c
   C. add.c
   D. update.c
   E. delete.c
   F. access.c
   G. DIcheck.c
INTRODUCTION

The Corpus Christi Army Depot (CCAD) is one of the many depots in the United States which provides logistical support for the Department of Defense. There are six directorates at this depot each with special functions. One of the functions of the Directorate of Information Management is to develop solutions for the unique problems encountered by the functional users of the Automated Data Processing System.

One problem commonly encountered was a slow response time when performing data file maintenance. The longest response times were being encountered while trying to update records currently stored on disk. Another problem was the large amount of error laden entries made by the functional users. Often the data entered was correct but the format was wrong. These errors occurred because users forgot the specific requirements of a particular file. Many users work on several files, and find it difficult to remember how many fields a certain file is supposed to have. A third problem occurred when users failed to put in the proper field separators when fields were empty. When a file is processed on the basis of the entry in the seventh field and there is not a seventh field, it causes difficulties.

The first step developing a solution to these problems was to analyze the current situation. The data files consisted of variable length records, consisting of variable length fields, and supposedly a fixed
number of fields. The bad response time associated with updating files came from the way in which the files were stored on disk, combined with the technique used for updating. When a record was entered without all the data being complete (some fields not containing any data) the field separators were entered without any data. For example:

"Tom Johnson:519 Parkwy:Denver, Colo"

might have been a complete entry. However, if the address was unknown at the time the record was added it should have looked like:

"Tom Johnson:::

However, it might have looked like:

"Tom Johnson:::

or even

"Tom Johnson".

Since the files were stored as variable length records whenever the file was updated it was necessary to lengthen the record, thus everything occurring after the updated record would have to be rewritten.

The preferred method of rewriting a file at CCAD is to copy the file to a new file until the place is reached where the new data should be added. The new data is written to the new file then the remainder of the old file is copied to the new file. After the copy is complete the new file is given the name of the old file. The disadvantage of this is, of course, the storage requirements to hold both the old and the new file are greater than with some other techniques. The advantage is that there is
little danger of loosing data. At most one record may be lost.

Although this method of copying files was safe, it was
time consuming to wait for the entire file to be rewritten on each
update. The combination of error prone input and file organ-
ization was making some data files difficult to work on.

The first thing to be changed had to be the rewritting of the
whole file on every update. The best way to avoid this was to set
a fixed length for each field and a fixed number of files for
each record. In this way only the updated record would be rewritten.

In this case even the solution caused some difficulties. The
method used to locate a record in the file was to use the "grep"
command. This command does a pattern string search of the entire
file, stopping when it finds a match. Naturally, the new fixed
length file would be longer since each field would have to be
capable of holding the maximum length entry for that field.
If the "grep" system command was to be used with the new file
organization, it would have to search a much longer file. This
could cause considerable degradation of performance on a long file.

The alternative solution chosen was to create a sorted key-
field directory which would be search using a binary search to lo-
cate a record. This would keep the number of compares to a minimum
and help produce a more reasonable response time.

The problem of incorrect formatting remained a perplexing problem
since it was desirable to keep the program general so that it could
be used with many data files. One solution was to make all the fields a standard size. This was considered to be impractical because of the great variation in field sizes of different files. An alternative was to have the program measure the size of the fields before starting. If it was possible to know that the field sizes were correct then the first record in the file could be used to determine the sizes of all the fields in a record. Since all records in a file would now be standardized to fixed lengths, the first record would have the same size fields as every other record in the file. It seemed a workable solution if only the user input problem could be solved. The only solution feasible was to take the problem of formatting away from the user and handle it in the program. With the program formatting all the fields in all the records, probability that the user would make a formatting error was reduced.
menu.c

The program "menu.c" creates the DI-AD-EME welcome screen as well as the first menu screen. The user may select any one of the four programs at this time, or he may choose to exit DI-AD-EME. The program menu.c is the parent process which causes a child process to be initiated as a result of the user's selection.

The program uses a switch set inside a while loop. The user is asked to choose one of the five alternative offered. If his choice is not one of those offered he is asked to select again. The result is placed in the variable "c" which is used as the test element in the switch. The choice made determines the character string to which "pgrm" will point.

The program begins with two include statements. The first statement causes the compiler to include the standard I/O library. The second include allows the program to handle interrupt signals. (It is not desirable for a program-abort interrupt in the child process, to terminate the parent process.)

The "main" function begins with the integer definitions of "status" and "more". "Status" will be used by the wait command in line 57 to suspend the calling process until a signal is received or until the child process terminates. The integer variable "more" is used for the test element in the while loop.

If the selection entered from the keyboard is a "c" then the character pointer "pgrm" is set to point to a string containing the
character pointer "pgrm" is set to point to a string containing the object code of the program "create.c". If the selection made is "a" then "pgrm" is assigned to point to the name of the object code of the program "add.c". When "u" is the choice "pgrm" will point to the object file of "update.c", and when "d" is chosen it will point to the object file of "delete.c".

The fork statement in line 48 causes the program "menu.c" to spawn the child process that "pgrm" names. Once the child process has finished executing the menu will be presented again and another child process may be spawned or the parent process may terminate, depending on the users selection.

The signal function (line 57) allows the delete key is entered from the keyboard to be ignored while the child process is running. The wait function (line 58) suspends the calling process until a signal is received from the child process. After the signal is received the status is reset to the default (line 59).

If the program is not terminated the while loop continues to be executed, spawning another child process. When the child process has finished the parent process may be terminated from the main menu when either the "x" is entered from the keyboard or when the delete key is pressed. (The delete key has been reset to the default by line 59).
create.c

The program create.c reads a file of variable length records and writes a new file of fixed length records and fields. The new file contains all of the information that was in the old file. Each field in the old file that was shorter than the measured necessary field size is padded out to the proper length with blanks in the new file. A directory is created for the new file consisting of a key field and an address offset of the record relative to the beginning of the file. The directory is sorted on the key field, so that a binary search can be used for record access.

The only preprocessor statement occurs in line one and causes the standard I/O library to be included. This is necessary for the input and output functions to perform correctly.

The program integer definitions begin on line 29 with the definition of the "field_max" integer array. The variables of the "field_max" array will be used to hold the maximum sizes of the fields. The variables of the array "vital_stor" will be used to hold the new maximum record size, the maximum number of fields, the total number of records in the file, the number of positions in the key field, and the length of a directory entry. The advantage in having this group of variables in consecutive array positions is that only one address will be passed to functions needing access to all this information.

The character type definitions begin on line 30 with "in_name" having 25 array positions which will be used to hold the name of the old file with variable length records. The name of the new file will be in the
The character type definitions begin on line 30 with "in_name" having 25 array positions which will be used to hold the name of the old file with variable length records. The name of the new file will be in the array "out_name" which may also have up to twenty-five characters. The directory name will be held in the character array "out_directory" and may have up to 30 characters.

The files to be used in this program are the old master file (variable length), the new master file (fixed length) and a keyed directory file for the new master file. Each file has a file pointer defined on line 36.

Creat.c is an interactive program which asks the user for the name of the variable length record file. The user types in the name of the file and the scanf statement in line 45 places the name into the character array called "in_name". Create.c then asks what the name of the new master file will be. This data is scanned and placed in the character array "out_file".

An attempt is made to open the old master file using the fopen function and the name in "in_file". The fopen is available to the program through the inclusion of the standard I/O library. The result of a successful open will be the return of unique file pointer (infile) which will be used throughout the program to identify the file. If the open is not successful then the NULL pointer is returned. (The value of the NULL pointer is identified in the standard I/O library.) The old master file is opened in the read mode since this file will not be written to by Create.c. If the attempt to open failed, the program terminates with
the failure message. If the file pointer returned is not NULL the program continues with a call to the function "file_counter" that reads the entire old master file. It places each incoming character into the integer variable "c". A character variable cannot be used to check for EOF. (EOF is the end of file marker and is defined in the standard I/O library as a minus one.) The variable "c", defined as an integer, can hold the minus one or any character. Following the initializations (lines 92 through 97) "file_counter" reads data into "c" and compares "c" to a field separator ("":") or a newline character ("\n"). The program considers all data coming in to be one of the following: field separator, new line character, or field characters. If "c" is neither colon nor newline, it must be a valid field character. When "c" is equal to a colon the counter for the number of fields is incremented (line 107) and the field size counter is set back to zero. When "c" is equal to a new line character the record counter is incremented (line 111), the counters for the field size and number are reset for the next record to zero and one respectively (lines 114 and 115). The maximum size of each field is kept and check after every increment of the counters. If the counter is larger than the maximum, the counter size become the new maximum (line 102 and 103). When the EOF value occurs in the variable "c" the while loop terminates and the appropriate values are stored in the vital_stor array variables. Control then returns to "main".

With the necessary field sizes now available "main" calls "write_fix_len" to write the new file and directory. At this time the user is queried as to the which field should be used as the key field. The name
of the new directory will be a concatenation of "dir" and key field number and the new master file name (line 210). A new file is opened for writing with the concatenated name.

The old master file is rewound and the lseek command is used to get the offset of the beginning of the first record in the file. The function "write _fix_len" begins reading data into the integer variable "c". The first possibility considered is that "c" might be a newline character. This variable is compared to a newline character. If the compare results in an equality the field counter is decremented, because newline characters are not counted as part of the field size of any field (line 224). The current field number is compared to the maximum number of fields allowable per record (stored in "vital _stor[2]"). If the current field number is not greater than the maximum number and the current field size is smaller than the maximum field size for this field, then the field is through 233). If the current field number is that of the key field then the directory entry is also padded out with blanks. The field separator is written to the master file and an end of string character is written to the directory file, if necessary. the remaining fields are padded out with blanks. If the key field number is reached at this time then a blank entry is entered in the directory (i.e. there was not any information in the key field of the old master file). After all the fields have been written to the new master file and the key field has been written to the directory, the offset for the record (stored in "mem_add") is written to the directory. The new line character follows the record address and completes the directory entry for the record.
The second possibility considered for "c" is that it might be a colon. In this case if the current field size is less than the maximum size for this field obviously the rest of the field must be padded out with blanks. If the current field is the key field then the entry into the directory must also be padded out with blanks. Each padded out blank increases the field size until the field size is equal the maximum size for this field. The field number is incremented by one (to be ready for the next field) and the field size is set to one.

The third and last possibility is that the character contained in "c" is a valid data character. The character is immediately written to the new master file. If the current field is equal to the key field the character is written to the directory file also.

The reading of characters continues until the end of file character is read. Once "c" is equal to EOF, the test condition of the while loop fails, and all the files are closed. Control returns to main.

The new master file has now been build and so has the directory. To allow the directory to be efficiently searched the directory will be sorted. The function "sort" is a shell sort which uses pointers. The directory entries are read into the integer array "intarray". "Vital_stor[3]" (the number of records in the file) is used as the loop control. Once all the records and addresses have been read in a message is displayed on the terminal screen that the program is "working". This is done so that the user know that the program is running.

The shell sort is entered at line 325 where "dist" (the distance of
compare) is set to half the number of records. During each iteration of
the outer loop the distance will be divided by two. The sort will continue
as until "dist" is not greater than zero. In the middle loop "i" is ini-
tially set to equal "dist" and is incremented by one with each iteration
of the middle loop until it is equal to the number of records in the file.
The inner loop does the actual comparing of the values at the address to
which the pointers refer. If the string compare returns a value greater
than zero the pointers are reverse. Pointer "ptarray[j]" would then point
to the value that "ptarray[j + dist]" used to point to, and vise versa.
The sort continues in the inner loop with all out of order records which
are compared being reversed. As "dist" is divided by two with each iter-
ation of the outer loop, the file becomes more ordered. Eventually the
distance is reduced to one on the last pass and the file is in order when
the pass ends.

The directory is printed out by the pointers in lines 345 through 350,
because the file is logically in order, rather than physically. The result
is a directory that is physically ordered on a key field. A message to
the user is sent to the terminal saying that the files are ready. A
"goodbye" lets him know the program is finished.
add.c

The program "add.c" provides a means by which files of fixed length records and fixed length fields that use colons as field separators may have records added. The records are added to the physical end of the master file, while the directory entries are inserted at the proper place in the index.

The character array "mas_name" will hold the name of the master file to which records will be added and "dir_cat" is the array which will hold name of the directory for the master file. The pointer which is the address of "dir_cat" is "dir_ptr".

The main function of add.c begins on line 11, with the necessary definitions necessary beginning on line 22. Although it is not desirable to explain the uses of every variable in the program at this time, it will enhance understanding to describe the more important ones. The first definition is "main_rec" which is the variable used to hold each character as it comes in from the master file and is used for comparisons. The character variable "dir_rec" performs the same function for the directory file. The function "getenv" is the function which obtains information about the terminal on which the user is working. The number of the field is picked from the screen in character form in the variables of "fieldchar". The integers of "mstr_flid_siz" contain the sizes of the fields in the master file. The integers of "dir_flid_siz" hold the sizes of the directory fields. The variable "more" is an integer switch to determine whether or not there are additional records to be added to the master file. The variables "num_of_m_fllds" and "num_of_d_fllds" contain
the number of fields in the master file and the number of fields in the
directory file. The two files to be used are "in_dir", the directory,
and "mastr_file", the master file.

The call to the signal function in line 28 allows the user to exit
the program at any time by pressing the delete key on the keyboard. When
"die" is called it causes the screen and the terminal to be reset to the
condition which existed before curses altered them. If "die" is not executed
to reset the terminal it may remain in the curses mode (set in line 30) and
respond function "initscr" in line 29. "Initscr" must be called to initial-
ize the screen routines before any of the screen routines may be used. It
initializes the terminal-type data and uses information gathered from "getenv".

Calling the "clear" function in line 31 causes the internal repre-
sentation (window) created by curses to be cleared. However, without
the refresh call in line 32 the physical screen seen by the user would
not be cleared. Refresh updates the physical screen to the current win-
dow view. The pointer "dir_ptr" is set to point to the first character
of the character array containing the string that is the name of the di-
rectory file. This is necessary since many instructions use the pointer
to an array rather than an array position.

The user is queried in lines 36 and 37 to determine to which file
he will be adding records. Accordingly, in line 38 the cursor is moved
to the correct position in the window, the screen is refreshed, and the
name of the file is put into "mas_name" character array. The string
contained in "mas_name" is then passed to the function "edit_str" which
edits the string for any unprintable characters the user might have
inadvertently entered.

The user is queried in lines 40 and 41 to determine which field of the master file will be used as key field. This number is put into the character string of "fieldchar" (line 44) because curses will only accept character data. The data in "fieldchar" is passed (line 43) to the function "edit_str" for editing.

The statements of line 47 through 54 are concerned with converting the character contained in "fieldchar" to an integer. DI-AD-EME will allow up to fifty fields to be used in a record it is possible for the data in "field_char" to contain a one digit number or a two digit number. Line 47 converts by casting, forced conversion, the character digit in "fieldchar[0]" to an integer, placing that integer in the variable "d_key".

The character digits all convert to consecutive numbers, with zero converting to forty-eight, one to forty-nine, two to fifty, etc. Line 49 subtracts forty-eight from "d_key" the result placed back into "d_key" will be a single integer digit. The second character digit is converted in the same manner in line 49. A check is made in line 50 to be certain that the converted data was actually a character digit. If so, the two digits (now integer) are combined and placed in "d_key" (line 54).

Line 56 attempts to open the master file with the name supplied by the user. If the open for reading was successful the program continues on, otherwise a message is written to the terminal telling the user the file could not be opened, and the function "die" is executed to reset the terminal. The user need not be aware that a directory is being used.

To accomplish this line 65 concatenates the prefix "dir" with the inte-
ger contained in "d_key" and the master file name. This concatenated string convention is used for naming the directory files throughout DI-AD-EME. Therefore, the user need not concern himself with the naming conventions of the directories he will be using. The string is placed in the array "dir_cat", and is used to open the directory in line 66. If the open attempt is unsuccessful, a message will be presented and "die" will be called.

The method used to determine the size and number of fields is to count the number of characters and fields in the first records of both the master and the index files. Colons are the only field separators allowed. The master file field sizes are measured by lines 78 through 90 with the results being placed in the variable array of integers, "mstr_fld_siz". The directory file field sizes are determined by similar code in lines 92 through 106. Both files are then closed.

The body of the loop used to add records begins on line 112. The "build_screen" function is called to build the window which allows the user to enter the data for the new record. After the data is entered the screen is cleared and refreshed and the user is queried as to whether more records will be added. The cursor is moved into position and "getst" places the response in "c". The response string is edited. If the answer is "y" the while loop is performed again, otherwise the "die" function is called and the program finishes.

The number fields per record and the size of each field were passed to "build_screen" in the variables of "num_of_m_flds" and "mstr_fld_siz". The two dimension character array "fieldab" defined in line 130 is used
to hold up to fifty fields each of which may be up to seventy characters in length.

The window for the header of the screen that the user will see is built by the function "add_header" which is called in line 135. "Build_screen" then puts to the screen the word "field" and the correct field number (1,2,3,...) followed by the maximum number of underscores (stored in masr_fld_siz) allowed for that field. This is accomplished by repetitive calls to the function "print_out_line" which creates the window and refreshes the screen (lines 244 through 261).

After the screen template has been created "build_screen" makes repetitive calls (line 161) to the function "pick_up_screen" which positions the cursor to the first underscore at the beginning of each line. When the user has pressed the return key the data entered on the screen on that line is put into the array "fieldab". The cursor is repositioned to the first underscore for the next field (see template A). The user will input the first eight fields on the first screen. Through a looping structure (lines 167 through 199) screens will continue to be created for the user until all fields (from one to fifty) have displayed. The user simply enters data over the underscores. If data is entered where there are no underscores, that data will not be put into the new record.

The function "edit_str" edits data coming from the terminal in the string (picked up by "pick_up_screen") for backspace characters, control characters, or any nonprintable characters. "Format" counts the number
of characters (line 283) in each string then copies that number of characters into the character array "charfield" and edits "charfield". Since curses performs no editing of it's own each string of characters may contain several back space characters. The user sees the cursor back up on the screen, and lines 286 through 291 adjust the input string to contain the same characters that the user sees on the screen. If the data was entered was shorter than the maximum length "format" pads out the field with blanks. If the data was longer than the maximum length "format" shortens the field to the proper length (line 301).

After all the fields have been formatted each of the character arrays of "fieldab" are concatenated into a single string (lines 206 through 211) in the character array "add_rec". The master file is then opened in the append mode (line 216) and the ftell command in line 223 ascertains the offset of the current location relative to the beginning of the file. This offset must be stored as a long integer and will become part of the directory entry. "Add_rec" is written to the file (line 225 and 226) and following the record a newline character is added.

The final function call in "build_screen" is to "write_indx_entry" which creates the directory entry for the record just added to the master file. A new file is opened called "NDX". As each entry is read from the old directory it is placed into the character array called "indx_entry" and compared to the key of the added record now placed in the character array "new_entry". If the index entry from the old index is less than the key of the new record, the old file entry is written to the new "NDX" file. If the key of the added record is less than the key
of the new record, then the key of the added record and its address are written to file.

Once the new entry is written to "NDX" the character array in "indx_entry" is written to the "NDX". The rest of the file may simply be copied from the old directory to "NDX". However, it is possible that the new entry is larger than every entry in the old directory and will never be written as a result of the compare. In that case the "last" switch is at zero when the end of file marker is encountered. After the loop terminates and "last" is tested, the new entry is written to file if it has not been written already.

Control is returned to main and the user is asked if there are more records to be added. The answer is edited by "edit_str" and compared to a "y". If there are not more records, "die" is called, otherwise the loop (line 111) is entered again.
SAMPLE ADD SCREEN TEMPLATES

TO ADD A RECORD

Please enter the information beside the appropriate field number, then press <Return>. If you wish to leave the field blank, press <Return>.

field 1: __
field 2: __
field 3: __
field 4: ________________
field 5: ________________
field 6: ________________
field 7: ________________

(first template)

TO ADD A RECORD

Please enter the information beside the appropriate field number, then press <Return>. If you wish to leave the field blank, press <Return>.

field 8: __
field 9: __
field 10: __
field 11: __
field 12: __
field 13: __

(second template)
update.c

The update.c program provides the a means by which the files created by DIADEME may be maintained. "Update.c" provides real time updating, so that if the user becomes confused as to whether or not he has made a certain update, the record can be accessed immediately and all updates made will have already been posted. All field sizes and record lengths are counted and stored by the program, thus the user need not remember any of the dimensions of the file. "Update.c" pads input fields which are too short with blanks and truncates fields which are entered too long.

To obtain the speed necessary for real time processing "update.c" uses a directory to the master file. Each entry in the directory contains the keyfield for a record in the file and the offset of the record address relative to the beginning of the file. The directory has been previously sorted on the keyed field. (If there is doubt as to the correctness of the order, this may be checked by DIADEME program "DIcheck".)

The input requirements are a file of fixed length records, each record consisting of a constant number of fixed length fields, and a keyed directory. All files must use colons as field separators. Duplicated records are acceptable, and the searches will find all records with the same key.

The program begins with the control statement (line 1) which causes the compiler to include the ctype.h file. The second include state-
ment causes the standard I/O library to be included. The third refers to the curses screen control, while the forth causes the interrupt signals to be made available. The fifth supplies the program with access to the system address pointers. The last include causes the system clock to be made accessible.

The function "die" is made global in line 6 because it is necessary to set the terminal back to the normal mode after curses has been used. Other global definitions are for the names of the master file and the directory file (line 16). The pointers to the character arrays containing the key fields will be called "ptr". The record to be updated will be placed in the character array "mastr_rec", after update it will be in "update_rec". Both arrays are defined to be 550 characters in length.

The "main" function begins on line 32. The character array "array" will hold the key field of the directories as they are read. The transaction log will record each update by using the pointer *audit defined in line 38. The structures timeval tp and timezone tzp will hold time as it is acquired from the system clock.

The first function call of "main" is to "signal", which assures that the response to an interrupt from the keyboard will be execution of the "die" function. In this manner the terminal is always reset to the standard mode for the terminal. "Initscr" is called (line 44) to initialize the screen prior the the creation of the windows.

The introduction routine is called on line 48 and begins on line
207. Before the window can be created the screen is cleared and refreshed. The user is queried (line 217) to obtain the name of the file he wishes to update. The "refresh" statement in line 222 causes the user to see the question. Without this statement the string of characters would only be added to the internal representation of the screen (window) to be used at some later time.

The user is also asked for the key field number (lines 238 through 240). A prefix of "dir" is concatenated to the field number and the master filename to yield the name of the directory. Thus the user does not need to remember the name of the directory nor even need to know that a directory is being used.

If the "fopens" (lines 226 and 256) are successful, the size of the directory entry is determined by the while loop (line 264) which counts each character of the first entry as it is read. Control then returns to "main".

Before the directory can be searched it is read into memory. The variable "total" is used to keep track of the total number of records in the file. The variables "total" and "more_rec" are initialize in lines 50 and 51 and will be used later for loop control.

The user is then asked for the key of the record he wished to update. This string is placed in the character array "search_key". At this time the string is not the same length as the key field entries, so a loop is set up to find the end of the string (line 65). It is padded to the proper length (line 69) or truncated if necessary (line 75).
Just prior to the search beginning the user is sent a message to advise him that the search is being done. On a large file it may be a several seconds. The key being used as a search key is sent to the screen, giving the user a chance to see if he has entered the correct key. If not, pressing the delete will take him back to the main menu where he can reenter the data.

The directory file is read (line 87) and the pointers are assigned (line 91). At the end of the file the loop test condition fails and the file is rewound. A pointer is assigned to the first character position of the array in which the search key is stored. This is done in preparation for using the compare string command which uses pointers. The binary search is used to find a match as the two pointers are used to compare the contents of the addresses they reference.

The binary search begins at line 107 where the variable "low" is set to zero and "high" is set to one less that the total number of records (now in "total"). With each iteration of the for loop the distance of compare is cut in half. If the result of the compare is stored in the variable "cond". If "cond" is not zero, either "low" or "high" will be reset. If "cond" is zero then a match has been found.

Once a match has been found the linear search (line 273) is called to find the first record with an entry equal to that of the search key. When the linear search determines which records is the first record, it displays the entire record on the screen, giving the user the opportunity to determine whether or not this was the record for which he was searching. If the users answer is negative then "linear search"
displays the second record with a key equal to that of the search key. The search continues in this manner until either the user's answer is positive or until there are no more records with the given key. If it is the latter then a message is sent to the screen to that effect. After the linear search is finished the master file is closed and control returns to the binary search in "main" (line 123). "Low" is set greater than "high" and the test condition fails on the next attempt.

If the search concludes without a match then "cond" is not zero. This means that there is not a record in the file with that key. The user is informed of that there are no records with the search key he gave and he is asked if he wishes to continue updating (line 140). The answer is stored in "more_rec". If the answer is "y" the condition of the test loop is met and control continues in "main" at line 52.

If the search ends and "cond" is zero, then a match was found. The user is queried to find out if he wishes to update the record (line 153). If the answer is "y", then the "update_screen" function is called, otherwise the "else" condition is not satisfied and control continues in the while loop at line 52.

"Update_screen" creates and displays the windows the user sees as templates for inputing the data for the records (line 322). The static character digits defined in line 327 and 328 will be used as field identifiers when put on the screen. The record is divided up by fields, with each field being put into an array, "field[a][b]". As each field is put into an array its size it counted in "field_siz"
and recorded in "mastr_fld_siz". The field size will be used to put underscore lines after the field entry is displayed on the screen.
(See sample update scene.)

"Update_header" is called (line 352) by "update_screen" to display a the instructions for using the screen template. The word "field" and a digit are displayed with each line to provide the user with a place to input each field. The integer field number must be converted to a character for curses. The conversion is done in lines 359 through 369. To print out each field line the print routine, "print_out_line" is called.

"Print_out_line" first print out the word "field", a colon, and a set of parenthesis. It then puts in the field number, which was passed from "update_header". The last character in the field is found by comparing each array position to a blank. If it is a blank the array count is backed up by one, if not, that number is saved as the last position in the array with a non-blank character. The array is printed out one character at a time until the last non-blank character has been printed. While the current count of printed characters is less than the field size, underscores are printed. In this way the user can see how many spaces he may use. (If he over runs the field on the screen, it is truncated to the correct length.) Control returns to "update_screen" (line 369) and the loop continues until all the fields have been printed or until six fields have been put to the screen, whichever is least.

The user is presented with the template screen and the cursor
is positioned, by a call to "pick_up_screen" (line 378) to a blank between a set of parenthesis. If he wishes to updated the screen he enters a "y". This causes a call to be made to the function "pick_up_new_field" (line 447) and the cursor jumps to the first position in the array. The user then begin typing in data. If he enters "n" the cursor jumps to the next set of parenthesis for the following field and again he may answer "y" of "n". When he has finished the first screen, the program continues to loop and create screens until all the fields have been displayed (lines 355 through 382).

Once the data has been entered it must be formatted to the field sizes of the file records. This is accomplished by a call to the function "format" in line 385. Format begins on line 488 and counts the characters in the field it was passed. It makes a copy of the field and stores the copy in the array "charfield". "Charfield" is test for valid characters. If backspace characters were used, this is accounted for (lines 500 through 506). The end result being is that "charfield" represents what the user saw on the screen, rather that the input string which was in "field". The edited string is copied back into field and control returns to "update_string" at line 387.

All of the fields are now formatted but they are still in separate arrays. The end of string character is placed in the array "update_rec" in the first position. Line 389 starts a loop which concatenates all the field strings into a single string. It works by searching the string found in "update_rec" until it locates the
end of string character. It then places the first character in the field array over the end of string character. It continues putting all the characters of the field into "update_rec" including the end of string character after the last character is copied from "field". The loop begins again with a search for the end of string character, which is now at the end of the copy of the first field. Once found the end of string characters is replaced with the first character of the second field, and the rest of the second field is copied on to the end of the first. The loop continues until all fields are copied into a single string. An end of string character is placed at the end and control returns to "main" at line 165.

The master file is opened again, the "fseek" command in line 173 uses the address found in the search to locate the correct position in the file. The "update_rec" is then written out to file. The audit log is opened (line 178) and the transaction is recorded, and the log is closed. The master file is closed, to force the update to be written to file.

The user is queried as to whether or not he wishes to update more records. This sets the loop control for the next pass. If the answer is "y" the loop begins again in line 52. If the answer is "n" the function "die" is called and the program terminates.
The program "delete.c" provides the user with a means to delete records from his view of the file without removing records from other views. This method allows a single master file to be viewed differently by users in separate areas. For example, a payroll file may need to contain a record of every employee who has worked this year. However, some employees who worked earlier in the year may no longer be employed yet their records need to be kept for year and processing. By deleting records only from the directory used for processing currently active employees, the record is not available for the monthly processing of paychecks, but remains in the master file for year end processing. This method is extremely useful in limiting redundancy.

The program begins with six include statements (their function described in the discussion of "add.c") followed by the definition of the function "die" which returns an integer. The signal statement in line 29 causes the function "die" to be executed if the delete key is entered from the keyboard.

The screen is initialized in line 30, followed by the mode being set, and the screen is being clear. The switch "more_recs" is set to "y" before going into the while loop that searches for the key of the record to be deleted.

The function "intro" is called (and begins on line 164) to get the name of the file from which the user wishes to delete records. If the file cannot be opened, the user is informed with a message sent to his terminal and the function "die" is called. If the open was successful
the user is queried for the number of the field that is to be used as key field. This field number is concatenated to the prefix "dir" and and a suffix which is the name of the master file. This name is used to open the correct directory file. If the directory file open is successful, the size of a directory entry is found by counting the number of characters first record of the directory file. Control returns to "main" (at line 36).

After the initializations are made the directory entries are read in and the pointers to the key fields are set (lines 37 through 53). The binary search begins (and is the same one used in the program "update.c". When a record with the key is found, the linear search is called (line 73) to locate the first record with the key.

If the key was not found, the user is informed and asked if there are other records he wishes to delete. If the search was successful the record is put to the screen by the linear search routine. Since there may be many records with a single key, the user is given the opportunity to view each record to determine whether or not it is the record he wanted to delete. A message is sent to the screen asking if this is the record to be deleted. A "n" answer causes the next record with the same key to be displayed and the user is questioned again as to whether or not this record selection is the one to be deleted. If the answer is "y" the function "delete_record" is called line 107.

The delete function works by reading in each directory entry with its offset address and comparing that address to the one that
is to be deleted. If the addresses are not equal, the entry is written to a new temporary directory. If the addresses are equal that entry is not written out to file. The rest of the file is simply copied from the directory to the temporary directory other. When the copy is complete the temporary directory is renamed to the name of the permanent directory. In this way there is no danger of data being lost should the system go down during a write to the file. After the file has been removed from the directory a transaction entry is made in the "DIAudit" file (the audit trail log) reflecting the delete and control returns to main at line 117.

Once in "main" the user is asked if there are more records to be deleted if so the loop begins again (line 37) otherwise the function "die" is called and the program terminates.
access.c

The program "access.c" uses the same file access methods as those used by the program "update.c". The difference is that this program uses the command line arguments passed to it by the calling instruction. The vector "argv" contains will contain four elements. The "argc" will be the number of arguments in the "argv" vector, in this case, it will be four. If it is not four a message is sent to the terminal advising the user of the proper usage.

Where the program "update.c" uses an interactive mode with the terminal, "access.c" receives its information through command line in the format of:

"programname filename keyfield recordkey".

If the record key is to contain spaces, then the entry must be as follows:

"programname filename keyfield "string entry"".

The entry sent as a string must be in quotations to be received as one key. The default is that every blank will be accepted as an argument delimiter.

"Access.c" uses a binary search to find the record (lines 87 through 102). Once the record a record is found with a matching key the record is displayed on the terminal screen (lines 138 through 145). The user is asked if this is the record for which he was looking. If not, the linear search (called by the binary search after the match was found in line 98) continues until the user responds with a "y" or until there are no more
matches in the file. Whenever the "y" is the answer the address offset of the file is returned to the calling program.
DIcheck.c

The program "DIcheck.c" provides a means whereby the serial continuity of the directory key field entries may be verified. The program begins with the definition of the character pointer array called "ptr". The directory name will be stored in the character array "dir_name" and will be known to the program by the file pointer "dir_file".

The first function call occurs in line 20 and is a call to the function "intro". "Intro" sends a character string to the screen asking the user for the name of the file he would like to have checked. The response is scanned and stored in "dir_name". An attempt is made while the size counter is incremented with each incoming character. When the colon is reached the counting ceases and the file is rewound. Control returns to main at line 23.

The pointer "ptr" is set to the address of the character array called "array" (line 22). The first entry is read into the array beginning array positions of "array" (line 26). As each entry, except the first entry, is read in it is compared to the previous entry. If the compare returns a value less than zero, the entries are out of order and a message is sent to the user advising him as to where the disorder begins. Progression continues through the file until the end is reached or records are found which are out of order.
SUMMARY & CONCLUSIONS

The first goal of this project was to improve the processing response time for real time updating of data files. This has been accomplished by decreasing the number of looks required located a record. In a file of 4000 records, the maximum number of looks required to locate the record using DI-AD-EME is 12 as compared to using the "grep" command which requires an average of 2000 looks to locate a record (provided the key field is the first field in the record).

There is a short wait before the first access may be made (approximately 10 seconds for 4000 records of 60 characters each) while the pointers are set. After the initializing takes place, a search for any record in the file requires less than 1 second to respond with a record which has the given key (if one exits in the file). However, if three records are updated during one terminal session the DI-AD-EME overall search response time is less than the "grep" response time for locating three records.

Although these improvements are appreciable, the greatest improvements were realized in the area of output. Formerly, writing a single updated record to file required thirty seconds or more (because the entire file was being rewritten and renamed). Using DI-AD-EME the write to file of any updated record to file requires approximately two seconds.

Less striking, but still measurable improvements were realized in the area of record deletes. Previously the deletes required the entire file, except for the deleted record, to be rewritten. Now, rewriting consists of rewriting the directory based on a single field. If there are ten fields this means only one tenth of the fields and their corresponding record
addresses are being rewritten. The time savings are still considerable.

The time savings gained while adding records to the file were similar to those experienced while deleting record since the procedures were basically equivalent. The format difficulties have been eliminated since the program now handles the formatting of all added and updated records.

The DI-AD-EME programs have accomplished the goals for which they were intended. The price for the speed gained has been paid through increased storage requirements. With storage becoming less expensive while labor becomes more expensive, the tradeoff seem to be a good one.
BIBLIOGRAPHY

