COST OF EXECUTION-TIME DATA-DEVELOPMENT FILE ACCESS

by

Joseph Kennedy

Memorandum No. ERL-453

June 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

COSTS OF EXECUTION-TIME DATA-DEPENDENT SECURITY

Joseph S. Kennedy

Electronics Research Laboratory and
Department of Electrical Engineering and
Computer Sciences
Computer Sciences Division
University of California
Berkeley, California

ABSTRACT

It is often convenient to allow access to a file in a data dependent manner. Access decisions made at translation time are sometimes inadequate, and postponement of data dependent access until execution time has previously been reported to be expensive both in time and space requirements. Some simple and reasonable assumptions in design of data dependent access methods facilitate efficient implementation and result in overhead costs which are quite small in almost all situations.

Research sponsored by the National Science Foundation Grant GJ-36475.

COSTS OF EXECUTION-TIME DATA-DEPENDENT SECURITY

Joseph S. Kennedy

INTRODUCTION

Conway, Maxwell, and Morgan [1] have shown that data dependent access methods are inadequate in many cases. Control over integrated data bases which include data of varying sensitivity sometimes requires access to the data which at least in part depends on the data itself. Thus, a file of personnel records may have many different access levels, and access to different levels may depend on the identity of the user. Certain records may be available to the accounting department, others to the company medical officer, and still others to management level personnel. The ability to allow access to a file but not to its entire contents obviates the need to maintain separate files with at least a partial duplication of data. Of course there will always exist the need for data independent access control since not every employee needs access of any kind. The question of whether to implement data dependent access control should be considered another data base design decision.

Translation-Time and Execution-Time Access Functions

Conway, Maxwell and Morgan have suggested the distinction between translation-time fetch and store functions (F_t and S_t respectively) and run-time fetch and store functions (F_r and S_r respectively). Each function takes as arguments a user identification u and a data item name d. At translation time, $F_t(u, d)$ checks a security matrix and decides whether or

not data independent access is granted. If so, normal object code is generated; if not, and if data dependent access is permitted, a call to $F_r(u, d)$ is generated in the object code. Otherwise, an abort is generated indicating lack of any access permission. (Similar actions are taken for store requests.) $F_r(u, d)$ makes the access check at run time and either returns the data item (if the data dependent check succeeds) or a null (if it fails). In this manner data independent access can be implemented with only a slight increase in translation time and no degradation in execution time. Data dependent access will, of course, result in loss of run-time efficiency proportional to the degree of data checking and the frequency of illegal access attempts (since $F_r(u, d)$ will have to generate and return a null record in that case).

Woodward and Hoffman [2] have pointed out several serious deficiencies in this method. Among them are (a) the necessity for all requests for file accesses to be known at translation time, (b) increased difficulty in ensuring integrity of object code after translation, and (c) the need to enforce retranslation of some, if not all, object code modules having access to a given data base if it is ever necessary to alter the data base access priviledges.

An Alternative Method - Modified System I/O Routines

It would be unfair to suggest that Conway et al. did not recognize some of these limitations. In fact, an alternative in which system I/O routines perform the F_r and S_r functions was discussed by them. In this case, F_t and S_t would be abandoned and all data accesses would be made at run time. Access control could then become translator independent, but performance degradation resulting from run-time access methods would become additional expense. They did not pursue this scheme.

Worst Case Costs

In the face of little empirical evidence, Woodward and Hoffman [2] conducted worst case studies to ascertain the feasibility of such an approach. Their results indicated that such run-time access might prove too expensive to garner wide acceptance. Worst-case degradations in run-time efficiency of 22% (for data independent access) to 146% (for data dependent access with a 100% access attempt failure rate) seemed to indicate that further work with the scheme described above was not warranted. However, their study was directed at worst case estimates, and no attempt was made to produce a "working solution" with, hopefully, more reasonable cost figures.

Another Implementation

We essentially implemented the F_r function with a slight modification to its definition given earlier. At every record access attempt, a data access flag was tested which could give three results. If negative, data independent access was not permitted and an abort routine implemented that could, although in this case did not, result in the erasure of all files written by the process making the access attempt. This is a simple way to implement run-time file access control where access privileges may change during the execution of the process. If the data access flag were zero, data independent access was permitted, and if positive, specific data dependent access was permitted. Before a more detailed description of this implementation can be given, a look at the file structure of the installation where the experiment was conducted is in order.

The system I/O routines of the Fortran library on the CDC 7600 at the Lawrence Berkeley Laboratories, University of California, Berkeley, were chosen for the measurements. A brief description of the sequence of events

resulting from the appearance of an I/O request (READ or WRITE) in the Fortran source code is given:

1. All local file names are identified by the compiler and assigned a File Environment Table (FET). This is completely independent of the file which will actually be referenced at run-time.
2. Each statement of the form READ (m,n) [input list] with k items in the input list generates an initial call to the system routine INPUTC which allocates a character buffer (we dealt only with files encoded in the CDC display code) tests and resets file status flags in the FET, and initializes the decoding routine KRAKER.
3. k calls to INPUTC result in the decoding (conversion of a character string to an internal representation. For example, the string "123" may be decoded to the real number 123, the integer 123, or simply as the symbol "123") of the input characters according to format n and transferral of the decoded data to addresses specified by the input list items. Note that a specific call to INPUTC may result in the decoding and movement of more than one data word.
4. A final call to INPUTC resets file status flags and clears the character buffer.

Therefore, we can summarize the activities in three phases: (1) initial call to the I/O routine for each record, (2) decoding of the data list items, and (3) a final status reset call.

In order to facilitate implementation, it was assumed that within a record, the "key", that data field on which the data dependent check is to be made, is in a known location for all records referenced by a particular READ statement. (All cost experiments were conducted using the input routines, although a similar treatment of output routines would suffice for S_r function implementation.)

Like most systems, BKY (the locally maintained operating system for the CDC 7600) utilizes a specific area of memory for communication between system I/O functions and user processes. This area is the File Environment Table in the user program space and consists of the 5 to 16 words allocated by the compiler for each file. It is within this area that three words were allocated for the data access control feature. (Of course, it would be necessary to move this area to a system protected region if any control were to be enforceable. It is assumed that this could be made readable to the user program with no penalty in access time.) One of the words was the Data Access Flag (DAF) and the other two provided upper and lower bounds for the record key. (These bounds could be real, integer, or character, etc.) More sophisticated means of access control and record key checking could be implemented with a slight increase in memory requirement. The values of the DAF and the key limits were set by the testing program; this function would pass to the operating system in an operational environment.

The Experiment

The experiment consisted of modifying INPUTC in order to implement the data access control method and executing a set of timing programs. The modification to INPUTC was made in the section of code executed when the first

call for the record was made. Just before a normal return, after the character buffer had been filled, the DAF was obtained from the FET and checked. If negative, a call to the system routine ABORT was initiated, indicating no access permission. If zero, a normal return to the calling program was initiated. If positive, indicating that data dependent access control was to be made, the record key was decoded from the character buffer and compared with the key limits in the FET. If within the given range, a normal routine was made. If out of the bounds set by the limit words, the character buffer was filled with a coded blank (octal 55 in the display code) and then the normal return given. It might seem like an undue restriction to require overwriting the entire record, but it is probably reasonable to assume that in most cases the data base could be organized to allow an all or none access to a record, since it can be arbitrarily small, down to one character. Modifications to INPUTC totaled an additional 13 words of coding.

Variable record size data was generated by another Fortran program and entered into the permanent (disk) file system for use by the timing programs. Each data file consisted of 100,000 records with a randomly generated key inserted in a given portion of the record. Record sizes of 5, 25, and 50 characters (excluding the record key) were used as the three test cases.

The timing programs were written in Fortran and consisted mainly of a timed loop executing a READ statement 100,000 times. Prior to this loop, the FET entries for the DAF and key limits, if necessary, were altered. With the DAF set to a positive value, the key limits could be altered to give a variable percentage of access attempt failures and subsequent overwriting, since the record keys were generated randomly over a fixed interval.

Each program was tested using 0%, 25%, 50%, 75% and 100% overwrite rates, with an additional two runs required for data independent access and the benchmark (the original INPUTC routine). Thus, seven runs on three sets of data were conducted.

Results

The results of the timing measurements are given in Figure 1. Both the actual timings and the performance degradation measured as the percent of CPU execution time increase are given for the three test cases.

Perhaps the most interesting result is the surprisingly low cost to implement data independent access control at run time. The high of 1.65% would seem like a small price to pay for the extra capability in those situations where this type of access control is desirable.

The sharp rise in degradation realized when the overwrite rate becomes high was an expected trend. What was not expected was the relatively small increase in view of [2]. Since the system may be considered to be designed poorly if accesses result in these high rates of failure, a somewhat lower figure would seem to give a more likely ceiling. In fact, if we may assume that access to a file in a data-dependent manner should result in a zero access failure rate, the figures for a 0% overwrite rate reflects an expected cost for such access control. The high of 28.27% appears to be an acceptable cost. Also, since these test cases were void of any computation, it might be more meaningful if computation time were added to the execution times and the resultant increases expressed as a percent. Figure 2 gives these interpretations for several 'computation times'. Now, costs seem to be even more attractive. For example, a job accessing 100,000 records of 50 characters each and requiring a total CPU time of approximately 30 seconds (20 seconds for input and 10 seconds of computation) would be paying

a cost of .56% for data-independent access of 5.38% for data-dependent access.

Conclusion

Execution time data-independent and data-dependent access control was implemented by modifying system I/O routines on a CDC 7600 and test cases run and timed to ascertain the cost involved. Experimental results indicate that this implementation would be cost effective and attractive for a wide range of applications where this capability is desired. A CPU overhead of around 1% - 2% for run-time data-independent access control is realized. Reasonable data base design should give cost increases for data dependent access control of 30% or below.

ACKNOWLEDGEMENTS

I wish to thank L.J. Hoffman whose discussions on the subject lead to these experiments, and to C.C. Bass, for insight in maintaining my perspective.

REFERENCES

[1] Conway, R.W., Maxwell, W.L., and Morgan, H.L., On the Implementation of Security Measures in Information Systems, Comm. ACM 15, 4, (April 1972), 211 - 220.

[2] Woodward, F. and Hoffman, L.J., Worst-Case Costs for Dynamic Data Element Security Decisions, Proc. ACM National Conference, San Diego, California, 1974.