Debugging PL/I Programs in the Multics Environment

by

Barry L. Wolman

Honeywell Information Systems

Cambridge Information Systems Laboratory

575 Technology Square

Cambridge, Massachusetts 02139

(617).491-6300

ABSTRACT

The problems associated with debugging a program written in PL/I are simplified in the Multics system because of a number of factors. PL/I is the standard language used for programming in Multics. Run-time features required by PL/I programs such as a stack, pointer variables, and a condition mechanism are all directly supported by the Multics system. The Multics PL/I compiler is complete, has few restrictions, and produces efficient object programs. A variety of powerful debugging commands make use of a run-time symbol table generated by the compiler thereby allowing the user to debug his program symbolically. Statistics about the operating characteristics of a program such as the time spent in a particular set of procedures or the cost of executing a given PL/I statement can be accurately determined.

Debugging PL/I Programs in the Multics Environment

by

Barry L. Wolman

Honeywell Information Systems

Cambridge Information Systems Laboratory

Cambridge, Massachusetts

INTRODUCTION

One of the popular misconceptions concerning PL/I is that programs written in PL/I are necessarily inefficient and hard to debug. Several years experience with the Multics PL/I compiler running on the Honeywell 645 has shown that in spite of the apparent complexity of the PL/I language, PL/I programs are easily debugged in the Multics environment, even by novice users who are newcomers to PL/I and are unfamiliar with the Honeywell 645. In most cases the user can debug his program symbolically without having to refer to a listing of the generated instructions or add debugging output statements to the program. This is due to a number of factors:

- the run-time environment provided by the system,
- the implementation of PL/I,
- the availability of a variety of powerful debugging facilities.

THE ENVIRONMENT

The use of PL/I as the principal tool for programming by users of Multics was envisioned at the very start of the project. Features which are required by PL/I such as a stack, pointer

variables, conditions, and a recursive call/return mechanism are all provided and are directly supported by the system hardware and/or software. Consequently, the basic Multics environment is ideally suited to the needs of PL/I Programs. In fact, nearly all of Multics itself is coded in PL/I and executes in this self-maintained environment.

The Multics system currently provides the user with a virtual address space of over 1000 segments of up to 65536 words each (some changes now in progress will increase the maximum size of a segment to 262144 words). Access to these segments is by means of PL/I pointer variables which contain a segment number, a word offset, and a bit offset. There is a direct correspondence between PL/I pointers and virtual addresses in Multics; PL/I pointer values may be loaded into the addressing registers of the 645 by a single machine instruction. An attempt to use a pointer whose value is the PL/I null pointer causes a condition to be signalled.

The PL/I stack is maintained for each user as a series of contiguous frames (block activation records) within a single segment. A register is dedicated by the system to point at the stack frame of the procedure being executed. Multics defines a system-wide standard call/save/return sequence which is relatively efficient. Stack frames can be obtained and released by executing a few instructions.

procedure segments in Multics are normally pure and sharable. Access to procedure and data segments is set by the

Multics access control commands and checked by the hardware at each instruction and data reference. If a user does not have appropriate access to a segment, or if any other error such as an attempt to divide by zero happens, a machine fault occurs. This fault is turned into a PL/I condition (e.g., "accessviolation" or "zerodivide") and is signalled by the PL/I condition mechanism. All but a few catastrophic errors are handled in this manner.

Multics provides a default error on_unit which is invoked if the user has not established an on-unit for a specific condition. In most cases, the default on-unit prints an appropriate error message (which may include information as to probable causes for the error) and calls the command processor to read a command from the user's input stream. The stack chain of calls leading up to the fault is preserved; in many cases the user's program can be restarted.

In Multics there is no real difference between a command and a program written by the user; both are PL/I procedures. Any program written in PL/I following command argument conventions may be invoked as a "command".

When the user types a command line of the form

```
edit alpha beta
```

the Multics command processor searches a specified set of directories for a procedure named "edit" and issues the equivalent of the PL/I statement

```
call edit("alpha","beta");
```

The procedures found in the system directories are the "commands"

and utility procedures normally available to Multics users. Since the user can change the search rules used by the system, he can tailor his own command set if he chooses.

THE IMPLEMENTATION

The implementation of PL/I in Multics is particularly complete and has few restrictions. The only omission of any consequence is tasking. The Multics implementation allows:

- arbitrary pointer qualification including chains of locators and use of functions as qualifiers,
- adjustable data with no restrictions. Arrays may have any number of adjustable bounds. Structures may have any number of adjustable members.
- operations on aggregates,
- functions which return values whose length or bounds are not known at the time the call is made, i.e., returns(char(*)) or returns((*) fixed bin).
- entry variables,
- full stream and record I/O,
- all data types including complex and decimal.

Since the implementation is so complete, the programmer does not have to worry about what features are or are not available to him. The ability to use the full language reduces the amount of code the user has to debug by increasing the amount of work handled by the run-time support system provided by the compiler.

The Multics PL/I compiler produces efficient object code, even when measured against the best efforts of experienced hand

coders using assembly language. The availability of a compiler which generates efficient programs greatly reduces the desire for users to want to switch to assembly language for reasons of efficiency. This is particularly important in Multics because of the richness of the machine instruction set (512 instructions and 64 types of address modification) and the complexity of the system environment from the view point of an assembly language coder.

Multics PL/I makes use of a separate "operator segment" which contains assembly language coding for about 50 commonly used functions such as string moving, complex multiplication, and the index operator, as well as tables of constants for masking, shifting, storing characters, etc. This segment is shared by all PL/I programs. Communication with the operator segment is by means of a work area in a standard position in each stack frame. The operator segment is entered by a short sequence of instructions which loads certain machine registers with parameters and then jumps directly into the operator segment at a known location. The use of the operator segment reduces the cost of PL/I programs by reducing their size and by reducing paging activity.

If a begin block or internal procedure block does not declare any automatic variables with adjustable bounds or sizes and can only be entered by first entering its parent block, then the block is said to be "quick". The Multics PL/I compiler does not use a separate stack frame for such blocks. Instead, they

share the stack frame of their parent block. The overhead of calling a quick block, exclusive of the cost of preparing the argument list, is only three instructions: one each at call, entry, and return. The cost of a quick procedure is also reduced because automatic storage in the parent block can be addressed directly.

The availability of a really inexpensive mechanism for internal procedures means that users can write them without having to worry about efficiency. The artifice of using label variables and goto statements to remove a block of code so that it can be executed efficiently from a number of places is not necessary.

The compiler makes no restrictions on the format of structures. This is important, since programmers can choose a structure description that is appropriate for the problem they are trying to solve without having to worry about its acceptability to the compiler. It is possible for a user to specify a structure which causes the compiler to generate very expensive accessing code. There are a few "common sense" rules users can follow if they are concerned about the efficiency of their programs.

Extensive error checking is done during compilation; there are nearly 500 possible error messages. Except for a few cases of multiple, related errors within a single statement the Multics PL/I compiler normally finds most errors in a single run. It is infrequent that a user will correct a set of source errors and

recompile his program only to receive another batch of error messages.

The listing generated by the compiler is designed to be printed by a high-speed line printer but is formatted so that items of interest to the user can be easily located by editing the listing segment on-line. The user can control the amount and level of detail of information placed in the listing.

DEBUGGING FACILITIES

Multics provides a number of special commands which aid user debugging. There is a powerful breakpoint debug command, a facility for tracing procedure calls, and tools which help the user determine the operating characteristics of his programs. There are several options that the user can specify when he uses the PL/I compiler to cause it to generate additional information for use by debugging commands.

The PL/I compiler and the system debug command cooperate to allow the user to debug his program symbolically. The compiler normally generates a run-time symbol table only if "get data" or "put data" statements are used in the source program. The compiler can be instructed to generate a "full" symbol table which includes all identifiers in the source program.

Each entry in the run-time symbol table describes an identifier in the user's program giving its name, storage class, location, size, bounds and other information needed to access the identifier. Information is available about the block in which the identifier is defined as well as its relationship to other

members of the structure to which it belongs.

The run-time symbol table facility is much more powerful than it needs to be just to support data directed I/O.

- Parameters, defined, and based variables can all be represented in the table. When a variable is declared based on a specific pointer, e.g., "dcl a based(p)", information is kept which allows the value of that pointer to be obtained at run-time.
- The size, offset, bounds, multipliers, or virtual origin of any identifier can be any arbitrary expression. This is necessary for the representation of based variables.
- References to identifiers in the user's program from data directed input or from requests to the system debugger need not be fully qualified. The same algorithm used by the compiler to resolve partially qualified names is also used by the support program which searches the run-time symbol table.

The run-time symbol table is generated at the end of the object segment and is shared by all users of the segment. If it is not used during execution, there is no overhead required to support it: the pages it occupies will not be brought into core memory; no code is required to initialize it. After the program has been debugged, the run-time symbol table can be eliminated from the object segment without having to re-compile it.

The compiler will also generate a "map" of the object program when a full symbol table is requested by the user. This map is a table, placed at the end of the object segment, giving information about the location in the object segment of each source statement. The availability of this table means that the user can refer to his object program by source line number, e.g., to set a breakpoint at a specific line number. Similarly, the system debugger can tell him the line number corresponding to a given location in the object program. In fact, the debugger can even print the source lines that correspond to the object location.

The command "debug" can be invoked at any time; for example, after an error condition has been signalled for which no on-unit exists. It may also be called directly from the user's program. It accepts requests from the user for actions such as examining some location in the virtual memory or printing a trace of the chain of calls in the user's stack. It is aware of the different PL/I data types, so variables in the object program may be displayed in the format appropriate to their type.

When a program has been compiled with a run-time symbol table, the user can refer to it symbolically, either with identifiers defined in the program or by the line number on which a statement begins. For example, if the user's program was dealing with a two-dimensional based array of integers, he could change one of the elements in the array by entering the request

```
p -> x(i+5,j-2) = 3
```

which takes the form of a PL/I style assignment. The values of "p", "i", and "j" would be obtained from the symbol table. Any of the identifiers in this example could be part of a structure.

The debug command has other features which let the more experienced user examine or alter the values in a machine register or display the status of the machine at the time a fault occurred. These facilities are not normally needed if a symbol table is available.

The debug command also lets the user set conditional or unconditional breakpoints in object segments. When a breakpoint is found during execution, the debug command is entered and if the associated condition is satisfied, a message is printed; at this point the user can enter requests. One of the actions available is to continue execution from the point of the break. There is a mode of execution available with debug which lets the user run his program one PL/I statement at a time.

An object program may have more than one break set in it; similarly, more than one program may have active breakpoints. Facilities are available in debug for listing and altering breaks. Setting a break involves changing the object program, so breakpoints remain active until explicitly removed by the user. Breakpoints should not be used when other users are sharing the segment.

There is an "escape" facility which causes debug to pass the line typed by the user to the Multics command processor instead of treating it as a request. This allows the user to invoke any

Multics command (or any of his own programs) without having to leave the debug command. He could, for example, run a special program to display the values of the static variables used by the program he is trying to debug.

The command "trace" lets the user monitor all calls to a specified set of external procedures. Trace modifies the standard Multics procedure linkage mechanism so that whenever control enters or leaves one of the procedures specified by the user, a debugging procedure is invoked. The arguments given to the debugging procedure by trace enable it to reference the arguments and return point of the procedure being called. The user can provide his own debugging procedures or he can use the one supplied as a default by the tracing package.

The action taken by the default trace debugging procedure is to print a message on the user's console whenever control enters or leaves one of the procedures being traced. There are a number of options which the user can specify to request such actions as printing the arguments (at entry, exit, or both) or stopping (at entry, exit, or both). The user can control the frequency with which the tracing message is printed, i.e., every 100 calls after the 1000th call. He can also specify the maximum recursion depth he wishes to see. The user can also request that the tracing message be printed only if the contents of some specified location in the virtual memory has changed.

The user may start tracing a procedure at any time, even it has already been executed. Tracing may be removed at any time;

subsequent calls of the procedure will execute normally. Any procedure which uses the standard Multics calling sequence may be traced without interfering with other users who may be sharing the segment.

The two debugging packages "debug" and "trace" which we have just discussed help the user find errors which prevent his program from running properly. There are another class of errors which are much harder to find. These are usually flaws in the program design (or perhaps in its implementation) which cause the program to run correctly but to take much longer to execute than it should. Simply locating the largest statement in the program or the biggest procedure is not sufficient to locate the causes of program inefficiency. Without detailed knowledge of program flow during execution, instruction counts alone are not much good.

The cost of executing a specified procedure, either for a single call or a total of many calls, can be determined by using the "meter" option of the trace command. The system clock counts in microsecond steps, so high resolution is possible.

Once a procedure has been found to be inefficient, its operating characteristics can be examined by re-compiling it with the PL/I "profile" option [6]. This option causes the compiler to generate in the internal static data area a table which contains an entry for each statement in the source program; the table entry contains information about the source line as well as a counter which starts out as zero. Each statement in the

program is modified to start with an instruction to add one to the counter associated with the statement.

After running a program compiled with the "profile" option, the user can determine the number of times each statement in the program was executed. The table entry contains the raw cost of the statement measured in instructions, so the user can determine both the absolute total cost for the statement as well as its relative cost compared to other statements.

The paging characteristics of a program can be measured by using the "page trace" facility. The Multics paging mechanism maintains a buffer for each user which records information about the last N page faults taken by the user's process (currently N = 256). A command is available which formats the information kept by the system.

DIFFICULTIES

As might be expected, there are problems associated with debugging PL/I programs in Multics. Most of these problems are minor and have the effect of requiring the user to know more about the internal workings of Multics than he might otherwise have to know.

The most difficult problem occurs when a program in the user's process commits an error so severe that the system cannot continue running the process. An example of such an error is using up the entire stack segment (perhaps because of unlimited recursion). When the system detects an error of this magnitude, it prints a message such as:

Fatal Process[illegible] Error

Out of bounds fault on user's stack

and creates a new process, thereby erasing all information about the old process.

This type of error can be very difficult to find, because no information is available to the user about where it occurred. Future versions of Multics will alleviate this problem by allowing the user to retain information about the old process. The system will also be changed to detect when the user is near the end of his stack; when this occurs, a special "stack" condition will be signalled.

EXPERIENCE

Most of the debugging techniques we have discussed have been used to aid the debugging of the PL/I compiler which is itself written in PL/I. The PL/I code generator, a program of 60,000 instructions of which half are modifications of previously debugged code, was written and debugged by the author in 15 months. The entire compiler (150,000 instructions of which two-thirds are new) was written and debugged in 6 - 8 man years.

REFERENCES

1. Organick, E. I., The Multics System: An Examination of Its Structure, M.I.T. Press (in press), Cambridge, Massachusettts and London, England.

2. Freiburghouse, R. A., "The Multics PL/I Compiler", AFIPS Conf. Proc. 35 (1969 FJCC), AFIPS Press, 1969, pp. 187-199.

3. Bensoussan, A., Clingen, C.T., and Daley, R. C., "The Multics

Virtual Memory", ACM Second Symposium on Operating System Principles, (October 20-22, 1969) Princeton University, pp. 30-42.

4. Daley, R. C., and Dennis, J. B., "Virtual Memory, Processes and Sharing in Multics", Comm. ACM 11, 5 (May 1968), pp. 306-312.

5. The Multiplexed Information and Computing Service: Programmers' Manual, M.I.T. Project MAC, Rev. 10, 1972. (Available from the M.I.T. Information Processing Center.)

6. Knuth, D., "An Empirical Study of FORTRAN Programs", Stanford University Computer Science Department Report No. CS-186.